# C++

## 11. HÉT – NODE JS NATIVE ADDONOK

# *Hogyan linkelhető C++ kód más nyelvek forráskódjához*

# Node.js-ről röviden

A Javascript (JS) 1995 óta (idestova 26 év óta) velünk van.

- Brendan Eich, a Netscape fejlesztője hozta létre abból a célból, hogy ne csak statikus weboldalak futhassanak a böngészőben
- 1996-ban sztenderdizálták először a nyelvet, ennek a sztenderdnek a neve azóta is ECMAScript. Ennek legfrissebb verziója a 2020-as ES11

A JS könnyen tanulható, felhasználóbarát nyelv. Viszont mivel mindössze 10 nap alatt készült(!) és rögtön igen népszerű lett, karbantartása és fejleszése a backward compatibility igények mellett nem volt könnyű.

- -> Az összes ügyes-bajos dolgával együtt a nyelv napjainkig egyfajta sztenderd maradt a böngészőben futó, dinamikusan interpretált programok létrehozásához

# Node.js-ről röviden

2009-ben Ryan Dahl gondolt egyet, és a Google V8 JS interpreterét felhasználva – azt egy libuv nevű könyvtárral és további API-val kiegészítve létrehozta a Node.js runtime-ot

- A “runtime” egy futtatási környezet, ami compilert / interpretert, valamint meghatározott programozási modellt / futtatási logikát biztosít végfelhasználói programokhoz
  - A javascript önmagában egy nyelv, amelyet korábban csak a böngészőkbe épített runtime tudott interpretálni. A Node.js egy a böngészőtől elvonatkoztatott runtime, amely szintén javascript kódot futtat, de saját programozási modellben

- A libuv könyvtár egy C könyvtár, amely platformfüggetlen módon megvalósít többek között:
  - egy ún. event loop-ot – ez folyamatosan nézi, hogy egy “task queue-ban” vannak-e még végrehajtandó feladatok (aszinkron callback-ek lényegében) és azokat végrehajtja
  - interfészt operációs rendszer szintű I/O műveletekhez
  - szálkezelést ezekhez az alacsonyszintű műveletekhez

# Node.js-ről röviden

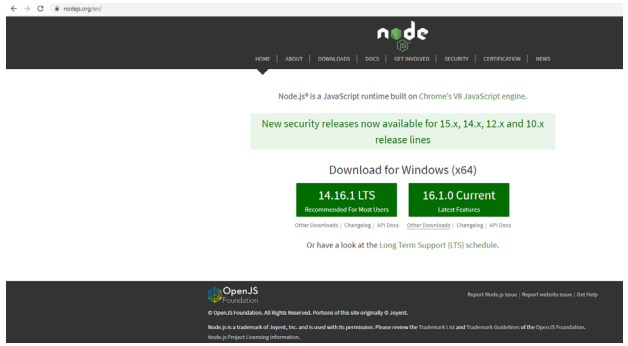
Beszéljünk világosan – mire jó a Node.js?

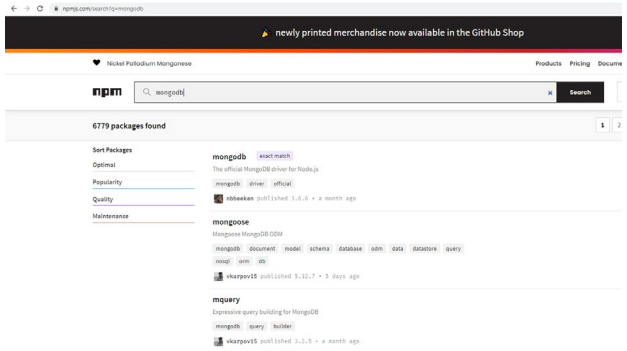
- Node.js segítségével böngészőtől függetlenül, “standalone” programként lehet Javascript (ECMAScript) kódot futtatni.
- Például nemcsak kliens, hanem szerver kód is készülhet ECMAScript nyelven. Ez azért jó, mert:
  - Nem kell más nyelv szerverre (PHP, Ruby), mint kliensre
  - A Node.js önmagában is egyedi és elegáns megoldást nyújt általános problémákra. Röviden: szemben a klasszikus megoldásokkal, ahol a szerver minden kéréshez új szálat kell hogy létrehozzon és ezeket szinkronizálja, a Node.js eseményvezérelt és nem-blokkoló logikával futtatja a kódot
    - Ha I/O műveletre várunk (pl. adatbázist kérdezünk le), ehhez nem kell külön szálat létrehoznunk és azt valamikor bevárnunk. Ehelyett a szerver fut tovább, és amikor a háttérkönyvtár (libuv vagy egyéb) végzett a lassú I/O művelettel, az általunk korábban megadott callback belekerül egy task queue-ba, amit az event loop ciklikusan feldolgoz
    - Az eredmény: a szerver garantáltan sosem várakozik, a nyelv pedig a programozó felé egy *látszólag* egyszálú működést biztosít – a programozó mindössze eseménykezelőket valósít meg.

# Node.js-ről röviden

A Node.js-hez kiterjedt ökoszisztéma is tartozik, melynek része pl. egy ún. package manager, a Node Package Manager (npm)

- Az npmjs.com oldalon böngészve láthatjuk, hogy mindenféle rendszerhez (akár szoftver, akár hardver) elérhető valamilyen Node.js API-t megvalósító csomag – legyen az adatbázis, wearable vagy IoT eszköz, stb. Emellett persze van rengeteg, a Node.js nyelvet kiterjesztő csomag is, pl. webserver létrehozását könnyítő csomagok (Express, Hapi, stb.) vagy csak a beépített API-t kiterjesztő csomagok (pl. fs-extra)
- Ha a nodejs.org oldalon letöltjük a Node.js runtime-ot, azzal együtt települ az npm mint command line program is



# Native addonok

Kanyarodjunk vissza a C++-hoz!

Modern fejlesztésnél nem ritka, hogy bizonyos funkciókat alacsonyabb szintű nyelven valósítanánk meg, másokat pedig magas szintű nyelven

- A hardverközelibb, sebességkritikus műveletek megvalósíthatóak C-ben vagy C++-ban (pl. 3D grafika, audio szintézis, fizikai folyamatok modellezése, neurális hálók vagy egyéb gépi tanulás modellek tanítása / inferenciája
- A felhasználóhoz közelebbi műveletek (felhasználói GUI, a program magasszintű logikája) megvalósíthatóak magasabb szintű nyelven, mint Python, Node.js, stb.

A Node.js dokumentációja szerint (https://nodejs.org/api/addons.html) a native addonok “dynamically linked shared objectek”, melyek C++ nyelven íródnak.

# Native addonok készítésének módjai

A natív addonok készítésének 3 módja létezik:

- A Node-API (a.k.a. NAPI) könyvtár használata
- A régebbi nan (Native Abstractions for Node) könyvtár használata (ezt váltotta fel a modernebb NAPI)
- Az alacsonyszintű V8, libuv és Node.js könyvtárak felhasználásával. Ez az opció kezdőknek nem ajánlott, mivel számos komponenst mélyen ismerni kell:
  - A v8.h headerben dokumentált V8 API-t, ami a Node.js által használt javascript implementáció (olyasmiket valósít meg, mint az objektumok létrehozása, függvényhívások az ECMAScript specifikációja szerint, stb.)
  - libuv könyvtárat, amely tartalmazza az event loop megvalósítását, valamint az alacsonyszintű “worker thread” kezelést
  - Belső node.js könyvtárakat, melyek szintén kiajánlanak bizonyos API-kat, amiket az addonok felhasználhatnak (pl. node::ObjectWrap osztály, amelyből származtatni lehet, ha azt szeretnénk, hogy adott osztályt bewrappelve felhasználhassunk, mint egy JS típust – így pl. new operátorral példányosíthatjuk)

# Native addonok NAPI-val

A NAPI nan-nal szembeni nagy előnye, hogy függetlenítve lett a Node.js alatti Javascript runtime-tól (V8-tól) – így ha később a V8 implementációja változik is, a NAPI-val létrehozott addonok (dll-ek) továbbra is működni fognak. A NAPI-t a Node.js projekttel együtt tartják karban.

- Korábban a nan használatakor lehettek olyan esetek, ahol a V8 frissülése miatt “eltört” egy-egy korábban működő addon (a kulcsszó itt az “ABI stability” – ahol az application binary interface arra utal, hogy milyen konvenció szerint sorrendezünk, méretezünk dolgokat egy bináris fájlban, pl. hány biten reprezentálunk adott típust, stb.)
- A NAPI támogatja C illetve C++ addonok létrehozását is. C++ addonok esetén egy további C++ wrapper modult kell használni, amely *node-addon-api* néven fut
  - A C++ wrapper felhasználóbarátabb, mert nem kell hibakódokat, illetve output argumentumként visszaadott paramétereket kezelni (a C API alacsonyabb szintű és ilyen “klasszikus” megoldásokat használ)
  - A következő fólián látható egy összehasonlító példa:

# Native addonok NAPI-val: C vs C++

C++ kód:

```
Object obj = Object::New(env);
obj["foo"] = String::New(env, "bar");
```

C kód, ami a bal oldallal ekvivalens:

```
napi_status status;
napi_value object, string;
status = napi_create_object(env, &object);
if (status != napi_ok) {
  napi_throw_error(env, ...);
  return;
}

status = napi_create_string_utf8(env, "bar", NAPI_AUTO_LENGTH, &string);
if (status != napi_ok) {
  napi_throw_error(env, ...);
  return;
}

status = napi_set_named_property(env, object, "foo", string);
if (status != napi_ok) {
  napi_throw_error(env, ...);
  return;
}
```

# Native addonok felépítése

Egy egyszerű példa így néz ki:

https://github.com/nodejs/node-addon-examples/tree/main/1_hello_world/node-addon-api

- a ***package.json*** tartalmazza a bindings és a node-addon-api függőségeket, valamint meghatározza, hogy a dll létrehozása node-gyp programmal történik (nem cmake-js-szel)

- a node-gyp programnak is van egy json-szerű konfigurációja, melynek neve ***binding.gyp***. Ez megmondja a node-gypnek, hogy milyen forrásfájlokból kell legenerálni az addont, milyen kapcsolókkal (kicsit olyan, mint egy makefile)

- emellett tartalmazza az addon legenerálásához szükséges C++ forrásfájlokat

- opcionálisan (ha ki szeretnénk próbálni az addont) tartalmazhat a projekt .js kiterjesztésű fájlokat is

# Native addonok: egyszerű példa

A példa futtatásához először a könyvtáron belül adjuk ki az “npm install” parancsot

- Ez telepíti az npm szerverről a package.json-ban megadott függőségeket, valamint a node-gyp segítségével lebuildeli a build/release mappába a dll-t (node-gyp rebuild)
- Mivel felhasználtuk a bindings csomagot, ez elrejti előlünk azt a komplexitást is, ami a dll létrehozásának helye és betöltése során ennek a helynek a megtalálásával kapcsolatos. Nem kell pl. ezt írni:

```
const addon = require('./build/Release/addon');
```

… amikor bizonyos rendszereken épp a Debug mappába kerül az addon.

- Az importálás és felhasználás ehelyett Javascriptből ilyen egyszerű (a bindings lényegében végigpróbálgatja az összes lehetséges helyét a dll-nek és az elsőt importálja, amit sikerül):

```
var addon = require('bindings')('hello');

console.log(addon.hello()); // 'world'
```

# Native addonok: egyszerű példa

A C++ kód pedig így néz ki:

```cpp
#include <napi.h>

Napi::String Method(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();
  return Napi::String::New(env, "world");
}

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  exports.Set(Napi::String::New(env, "hello"),
              Napi::Function::New(env, Method));
  return exports;
}

NODE_API_MODULE(hello, Init)
```

Itt némi magyarázat szükséges:

- A *NODE_API_MODULE* egy makró, amely a modul nevét és egy regisztráló függvény nevét várja
- A regisztráló függvény egy “exports” objektumot ad vissza, amelybe *Set()*-tel beállíthatóak kulcs-érték párként, hogy miket exportálunk
  - Itt most egy “hello” nevű modult készítünk, amely exportál egy “*hello()*” nevű függvényt, amit a Method függvény valósít meg
- Van itt egy adag boilerplate – pl. az *Env* argumentumot legtöbb NAPI függvénynek meg kell adni – ezt a Node.js futtatási környezet adja át amikor meghívjuk a NAPI függvényeket, és az aktuális névteret / környezetet reprezentálja.
- A *CallbackInfo* argumentum tartalmazhatja pl. a felhasználó által átadott paramétereket, melyek indexálással érhetőek el (mint egy tömbnél), és melyeknek a száma lekérdezhető a *Length()* metódussal

# Native addonok

Itt egy buta példa, amely argumentumok átadását mutatja be, a CallbackInfo argumentum segítségével. Láthatjuk, hogy a JS adattípusok és a C++ adattípusok között adott esetben konverzió szükséges (marshalling)

```cpp
#include <napi.h>
#include <string>

Napi::String Method(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();
  std::string result = "world";
  if (info.Length() > 0) {
    for (int i = 0; i < info[0].As<Napi::Number>().Int32Value(); i++) {
        result += '.';
    }
  }
  return Napi::String::New(env, result);
}

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  exports.Set(Napi::String::New(env, "hello"),
              Napi::Function::New(env, Method));
  return exports;
}

NODE_API_MODULE(hello, Init)
```

```js
var addon = require('bindings')('hello');

console.log(addon.hello()); // 'world'

console.log(addon.hello(3)); // 'world...'

console.log(addon.hello(1)); // 'world.'

console.log(addon.hello(-1)); // 'world'
```

```
c:\Users\Adamc\Desktop\C++-Advanced\node-addon-examples\1_hello_world\node-addon-api>node .
world
world...
world.
world
```

# Native addonok: Promise-ok

Ha egy számítás hosszú ideig tart és külön szálba szervezzük ki, használjunk promise-okat

(Ha visszaemlékszünk, C++-ban is létezik ilyen: egy promise objektumot adunk át a meghívott függvénynek, ami beállíthat rajta egy eredményt. Erről pedig egy külön future objektumon keresztül tud a hívó oldal értesülni, melyet korábban a promise objektumból kérdezett le)

A Promise mint típus egy ideje már az ECMAScriptnek is része, így a Node.js-ben is benne van és a NAPI is lehetővé teszi, hogy promise-okkal operáljunk.

Ha egy native addonban hosszadalmas műveletet kell végrehajtanunk, megtehetjük, hogy új szálat indítunk (akár direct módon, akár async taszk segítségével), és egy ***Napi::Promise::Deferred*** típusú objektumot használunk fel a hívó féllel való kommunikációhoz.

- A feladat egyszerűsítése érdekében viszont (hogy ne kelljen bajlódnunk az adatok megosztásával a JS event loop és a worker szálak között), célszerű mindezt egy a ***Napi::AsnycWorker*** absztrakt osztály kiterjesztésével megvalósítanunk! (ez kicsit olyan, mint az std::async, csak a Node.JS – worker thread világra adaptálva)

# Native addonok: Promise-ok

Nézzük a saját worker osztályunkat apránként!

```cpp
#include <napi.h>
#include <string>
#include <thread>
#include <chrono> // std::chrono::seconds miatt

class MyAsyncWorker : public Napi::AsyncWorker {
public:
  static Napi::Value Create(const Napi::CallbackInfo& info) {
    if (info.Length() != 1 || !info[0].IsNumber()) {
      Reject(info.Env(), "invalid input - expects a single integer value"); // Reject protected static metodus
    }

    int timeToSleep = info[0].As<Napi::Number>().Int32Value();

    MyAsyncWorker* worker = new MyAsyncWorker(info.Env(), timeToSleep);
    worker->Queue();
    return worker->deferredPromise.Promise();
  }
protected:
  void Execute() override {
    std::this_thread::sleep_for(std::chrono::seconds(timeout));
    returnValue = "slept for " + std::to_string(timeout) + " seconds";
  }
```

A *Create()* nevű statikus metódus szerepe:

- az argumentumokat ellenőrzi,
- az argumentumokat átkonvertálja napisról natív C++-os típusokra
- végrehajtási sorba helyezi a példányosított worker objektumot
- visszaadja a promise-t a hívó félnek (az eredménye majd ebbe kerül)

Emellett látjuk, hogy az *Execute()* metódus felüldefiniálása szükséges. Ez végzi el a külön szálon futó végrehajtást.

# Native addonok: Promise-ok

Nézzük a saját worker osztályunkat apránként!

```
  static Napi::Value Reject(Napi::Env env, const char* msg) {
    Napi::Promise::Deferred failed = Napi::Promise::Deferred::New(env);
    failed.Reject(Napi::Error::New(env, msg).Value());
    return failed.Promise();
  }

  virtual void OnOK() override {
    deferredPromise.Resolve(Napi::String::New(Env(), returnValue));
  }

  virtual void OnError(const Napi::Error& e) override {
    deferredPromise.Reject(e.Value());
  }
private:
  MyAsyncWorker(napi_env env, int timeToSleep) :
    Napi::AsyncWorker(env),
    timeout(timeToSleep),
    deferredPromise(Napi::Promise::Deferred::New(env)) { }

  int timeout;
  std::string returnValue;
  Napi::Promise::Deferred deferredPromise;
};
```

A *Reject()* nevű statikus metódus segít megkímél bennünket a boilerplate-től, segítségével rejectelő promise-t lehet visszaadni

Az *OnOK()* és *OnError()* metódusok szintén virtuális metódusok az absztrakt ősosztályban, melyek felüldefiniálandóak! Ezek állítják be a megfelelő eredményt

Az osztály konstruktora privát, mivel a statikus *Create()* metódussal példányosíthatjuk csak. A konstruktor inicializálja az osztályszintű változókat, melyekre szüksége van a többi metódusnak.

# Native addonok: Promise-ok és callbackek

Az exportálás így néz ki – a kód ezen részében nincs újdonság:

```
Napi::Object Init(Napi::Env env, Napi::Object exports) {
  exports.Set(Napi::String::New(env, "returnAfterNSecs"),
              Napi::Function::New(env, MyAsyncWorker::Create));
  return exports;
}

NODE_API_MODULE(hello, Init)
```

Végezetül a JS oldalon így használhatjuk a DLL-t:

```
var addon = require('bindings')('hello');

let time = 1;
let timePrintInterval = setInterval(() => {
  console.log('time: ', time)
  time++;
}, 1000)


addon.returnAfterNSecs(5).then(returnedVal => {
  console.log('received after 5 seconds: ', returnedVal)
})

addon.returnAfterNSecs(8).then(returnedVal => {
  console.log('received after 8 seconds: ', returnedVal)
  clearInterval(timePrintInterval)
})

let mySetTimeout = (timeoutlen, cb) => {
  addon.returnAfterNSecs(timeoutlen).then(() => {
    cb()
  })
}

mySetTimeout(12, () => {console.log("Twelve seconds have passed")} )
```

**Megjegyezzük, hogy natív addonból nemcsak adatot adhatunk vissza a JS oldal felé, hanem meg is hívhatunk egy Napi::Function típussal reprezentált JS függvényt!**

- De ha ezt nem a main threadből tesszük, akkor ehhez a *Napi::ThreadSafeFunction* vagy *Napi::TypedThreadSafeFunction* osztályokat kell felhasználni. Ezek a típusok szálbiztos módon kommunikálnak az addon main threadjével, így az meghívhatja a nevükben a megfelelő JS függvényeket.

# Native addonok – Objektumok becsomagolása

Végezetül nézzünk egy olyan példát, ahol a C++ oldal egy objektumot hoz létre, melyet JS oldalról fel lehet használni

- Az előző példában is valami hasonlót csináltunk, csak az objektumhoz implicit módon – egy statikus függvényen keresztül fértünk hozzá
- Ez nem maga az objektum volt, hanem egy statikus függvény ami az objektumot példányosította, majd visszaadott egy promise-t

Ha azt szeretnénk, hogy JS oldalon példányosítani lehessen egy C++ oldalon definiált objektumot, majd hívogatni lehessen a metódusait, egy Napi::ObjectWrap<T> nevű osztályt kell kiterjesztenie a saját osztályunknak (ahol T a saját osztályunk típusa, amit átadunk template argumentumként)

- Igen, ilyet lehet csinálni, hogy a szülőosztály template argumentuma a gyermek osztály típusa

E mellett az osztálynak fix szintaxis szerint be kell regisztrálnia egy Class property and descriptoron kereszül, hogy milyen metódusokat ajánl ki az interfészén.

# Native addonok – Objektumok becsomagolása

Tegyük fel tehát, hogy JS oldalon egy ilyen interfészt szeretnénk:

```
var addon = require('bindings')('addon');

var obj = new addon.MyObject(10);
console.log( obj.plusOne() ); // 11
console.log( obj.plusOne() ); // 12
console.log( obj.plusOne() ); // 13

console.log( obj.multiply().value() ); // 13
console.log( obj.multiply(10).value() ); // 130

var newobj = obj.multiply(-1);
console.log( newobj.value() ); // -13
console.log( obj === newobj ); // false
```

# Native addonok – Objektumok becsomagolása

Ilyenkor az exportálás egy kicsit más szintaxissal történik, de csak látszólag. Az exports objektumot most átadjuk egy statikus Init() metódusnak, és ez fogja beregisztrálni az objektum leíróját:

```cpp
#include <napi.h>
#include "myobject.h"

Napi::Object InitAll(Napi::Env env, Napi::Object exports) {
  return MyObject::Init(env, exports);
}

NODE_API_MODULE(addon, InitAll)
```

Nézzük emelett az objektum header-jét (a kiterjesztés a lényeges, valamint az Init() static metódus – a többi ismerős):

```cpp
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <napi.h>

class MyObject : public Napi::ObjectWrap<MyObject> {
 public:
  static Napi::Object Init(Napi::Env env, Napi::Object exports);
  MyObject(const Napi::CallbackInfo& info);

 private:
  Napi::Value GetValue(const Napi::CallbackInfo& info);
  Napi::Value PlusOne(const Napi::CallbackInfo& info);
  Napi::Value Multiply(const Napi::CallbackInfo& info);

  double value_;
};

#endif
```

# Native addonok – Objektumok becsomagolása

Az Init() statikus metódus implementációja így néz ki:

```
Napi::Object MyObject::Init(Napi::Env env, Napi::Object exports) {
  Napi::Function func =
      DefineClass(env,
                  "MyObject",
                  {InstanceMethod("plusOne", &MyObject::PlusOne),
                   InstanceMethod("value", &MyObject::GetValue),
                   InstanceMethod("multiply", &MyObject::Multiply)});

  Napi::FunctionReference* constructor = new Napi::FunctionReference();
  *constructor = Napi::Persistent(func);
  env.SetInstanceData(constructor);

  exports.Set("MyObject", func);
  return exports;
}
```

Ez a metódus egy függvényt regisztrál be az exports objektumba, melynek neve az osztály neve.

A függvény típusa Napi::Function (ez egy JS oldalon meghívható függvényt reprezentál), és amikor meghívjuk, lényegében egy osztályt definícióját kapjuk vissza

- De ahhoz, hogy ezt a függvényt újra meg újra meg tudjuk hívni, az Init metódusnak perzisztenssé kell tennie (ne törölje ki a garbage collector, miután az Init() lefutott.
- Ezért adjuk át egy FunctionReference pointernek a dinamikus memóriában
- Emellett hozzá társítjuk a pointert az adott addon instance-hez. Ez azért jó, mert amikor az addon-t felszámoljuk, ezt a dinamikus memóriaterületet is fel kell szabadítani.