دليلك إلى Node.js

تأليف
Flavio Copes
ترجمة
علا عباس
عبد اللطيف ايمش

دليلك إلى Node.js

دليل مبسط للمبتدئين لتعلم أساسيات بيئة نود جي إس Node.js

Book Title: Node.js Handbook	**اسم الكتاب**: دليلك إلى Node.js
Author: Flavio Copes	**المؤلف**: فلافيو كوبس
Translator: Ola Abbas - Abdullatif Eymash	**المترجم**: علا عباس - عبد اللطيف ايمش
Editor: Ayat Alyatakan - Jamil Bailony	**المحرر**: آيات اليطقان - جميل بيلوني
Cover Design: Sirin Diraneyya	**تصميم الغلاف**: سيرين ديرانية
Publication Year: 2024	**سنة النشر**:
Edition: 1.0	**رقم الإصدار**:

بعض الحقوق محفوظة - أكاديمية حسوب.

أكاديمية حسوب أحد مشاريع شركة حسوب محدودة المسؤولية.

مسجلة في المملكة المتحدة برقم 07571594.

https://academy.hsoub.com

academy@hsoub.com

Copyright Notice

The author publishes this work under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

This license is acceptable for Free Cultural Works.
The licensor cannot revoke these freedoms as long as you follow the license terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
Read the text of the full license on the following link:

إشعار حقوق التأليف والنشر

ينشر المصنِّف هذا العمل وفقا لرخصة المشاع الإبداعي نَسب المُصنَّف - غير تجاري - الترخيص بالمثل 4.0 دولي (CC BY-NC-SA 4.0).

لك مطلق الحرية في:

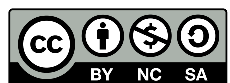
- المشاركة — نسخ وتوزيع ونقل العمل لأي وسط أو شكل.
- التعديل — المزج، التحويل، والإضافة على العمل.

هذه الرخصة متوافقة مع أعمال الثقافة الحرة.
لا يمكن للمرخِّص إلغاء هذه الصلاحيات طالما اتبعت شروط الرخصة:

- نَسب المُصنَّف — يجب عليك نَسب العمل لصاحبه بطريقة مناسبة، وتوفير رابط للترخيص، وبيان إذا ما قد أُجريت أي تعديلات على العمل. يمكنك القيام بهذا بأي طريقة مناسبة، ولكن على ألا يتم ذلك بطريقة توحي بأن المؤلف أو المرخِّص مؤيد لك أو لعملك.
- غير تجاري — لا يمكنك استخدام هذا العمل لأغراض تجارية.
- الترخيص بالمثل — إذا قمت بأي تعديل، تغيير، أو إضافة على هذا العمل، فيجب عليك توزيع العمل الناتج بنفس شروط ترخيص العمل الأصلي.

منع القيود الإضافية — يجب عليك ألا تطبق أي شروط قانونية أو تدابير تكنولوجية تقيد الآخرين من ممارسة الصلاحيات التي تسمح بها الرخصة.
اقرأ النص الكامل للرخصة عبر الرابط التالي:

https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode



The illustrations used in this book is created by the author and all are licensed with a license compatible with the previously stated license.

الصور المستخدمة في هذا الكتاب من إعداد المؤلف وهي كلها مرخصة برخصة متوافقة مع الرخصة السابقة.

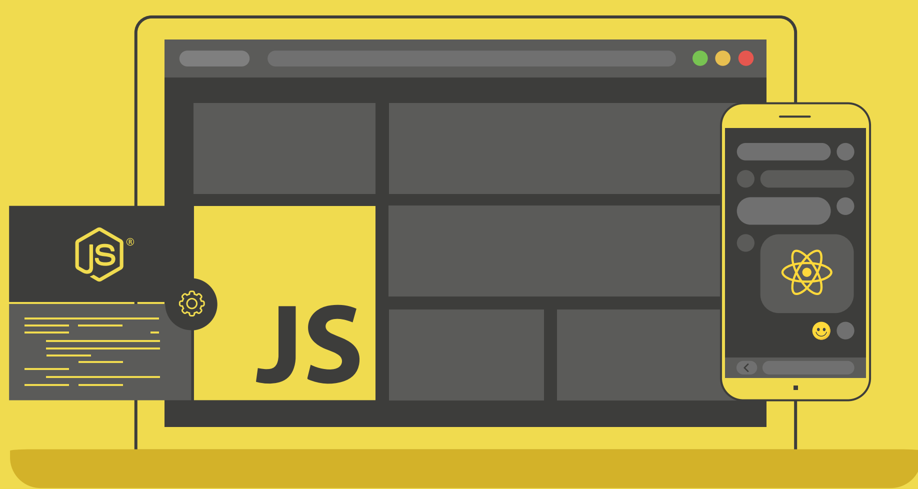
عن الناشر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العـربي عـبر توفـير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخـرى، بالإضـافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهلـه حـتى دخول سوق العمل.

حسوب شركة تقنية في مهمة لتطوير العالم العـربي. تبـني حسـوب منتجـات تركِّـز على تحسـين مسـتقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة تطوير التطبيقات باستخدام لغة JavaScript



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

المحتويات باختصار

تمهيد 16

1. مقدمة إلى Node.js 17

2. استخدام الوضع التفاعلي في Node.js 33

3. مدير الحزم npm في Node.js 43

4. كيفية تنفيذ الدوال داخليا ضمن Node.js 73

5. البرمجة غير المتزامنة في Node.js 86

6. التعامل مع طلبيات الشبكية في Node.js 98

7. التعامل مع الملفات في Node.js 118

8. تعرف على وحدات Node.js الأساسية 132

جدول المحتويات

تمهيد **16**

1. مقدمة إلى Node.js **17**

1.1 أفضل مميزات Node.js 17

1.1.1 السرعة 17

1.1.2 البساطة 18

1.1.3 تستعمل JavaScript 18

1.1.4 تستخدم محرك V8 18

1.1.5 منصة غير متزامنة 18

1.1.6 عدد هائل من المكتبات 19

1.2 مثال عن تطبيق Node.js 19

1.3 أدوات Node.js وأطر عملها 20

1.4 تاريخ موجز عن Node.js 21

1.5 كيفية تثبيت Node.js 22

1.6 ماذا عليك معرفته في JavaScript لاستخدام Node.js 22

1.7 الاختلافات بين Node.js والمتصفح 24

1.8 محرك V8 25

1.8.1 محركات جافا سكريبت الأخرى 25

1.8.2 السعي إلى الأداء الأفضل 25

1.8.3 التصريف Compilation 25

1.9 تشغيل تطبيق Node.js والخروج منه 26

1.10 متغيرات البيئة: الفرق بين التطوير والإنتاج 28

1.11 استضافة مشاريع Node.js 29

1.11.1 نشر التطبيق في نفق محلي 30

1.11.2 نشر التطبيقات دون أي ضبط 30

1. Glitch 30

1. Codepen 31

1.11.3 الخيارات عديمة الخوادم Serverless 31

1.11.4 المنصة على أساس خدمة PAAS 31
1. Zeit Now 31
1. Heroku 31
1. Microsoft Azure 31
1. Google Cloud Platform 32
1. Vercel 32
1.11.5 خادم خاص افتراضي VPS 32
1.11.6 خادم حقيقي 32
2. استخدام الوضع التفاعلي في Node.js 33
2.1 استخدام الوضع التفاعلي REPL 33
2.1.1 استخدام الزر tab للإكمال التلقائي 34
2.1.2 استكشاف كائنات JavaScript 34
2.1.3 استكشاف الكائنات العامة 35
2.1.4 المتغير الخاص _ 35
2.1.5 الأوامر ذوات النقط 35
2.2 تمرير الوسائط من سطر الأوامر إلى Node.js 36
2.3 إرسال تطبيق Node.js المخرجات إلى سطر الأوامر 37
2.3.1 طباعة المخرجات باستخدام الوحدة console 37
2.3.2 مسح محتوى الطرفية 38
2.3.3 عد العناصر 38
2.3.4 طباعة تتبع مكدس الاستدعاء 39
2.3.5 طباعة الزمن المستغرق 40
2.3.6 مجرى الخرج القياسي stdout والخطأ القياسي stderr 40
2.3.7 طباعة المخرجات بألوان مختلفة 40
2.3.8 إنشاء شريط تقدم في الطرفية 41
2.4 قبول المدخلات من سطر الأوامر 41
3. مدير الحزم npm في Node.js 43
3.1 إدارة تنزيل الحزم والمكتبات والاعتماديات 44
3.1.1 تثبيت جميع الاعتماديات Dependencies 44

3.1.2 تثبيت حزمة واحدة 44

3.1.3 مكان تثبيت npm للحزم 45

3.1.4 كيفية استخدام أو تنفيذ حزمة مثبتة باستخدام npm 46

3.1.5 تحديث الحزم 47

3.1.6 إدارة الإصدارات وسرد إصدارات الحزم المثبتة 50

3.1.7 إلغاء تثبيت حزم npm 51

3.2 تشغيل مهام وتنفيذ سكربتات من سطر الأوامر 52

3.3 الملف package.json نقطة ارتكاز المشروع 53

3.3.1 معمارية الملف package.json 53

3.3.2 خاصيات الملف package.json 57

ا. name 57

ب. author 57

ج. contributors 58

د. bugs 58

ه. homepage 58

و. version 59

ز. license 59

ح. keywords 59

ط. description 59

ي. repository 59

ك. main 60

ل. private 60

م. scripts 60

ن. dependencies 61

س. devDependencies 61

ع. engines 62

ف. browserslist 62

ص. خصائص خاصة بالأوامر 62

3.3.3 إصدارات الحزم 63

3.4 الملف package-lock.json ودوره في إدارة الإصدارات 63
3.5 قواعد الإدارة الدلالية لنسخ الاعتماديات 67
3.6 أنواع الحزم 69
3.6.1 الحزم العامة والحزم المحلية 69
3.6.2 الاعتماديات الأساسية واعتماديات التطوير 70
3.7 أداة تشغيل الشيفرةnpx 70
3.7.1 تشغيل الأوامر المحلية بسهولة 71
3.7.2 تنفيذ الأوامر دون تثبيتها 71
3.7.3 تشغيل شيفرة باستخدام إصدار نود Node مختلف 72
3.7.4 تشغيل أجزاء شيفرة عشوائية مباشرة من عنوان URL 72
4. كيفية تنفيذ الدوال داخليا ضمن Node.js 73
4.1 حلقة الأحداث event loop 73
4.1.1 مدخل إلى حلقة الأحداث 73
4.1.2 إيقاف حلقة الأحداث 74
4.1.3 مكدس الاستدعاءات call stack 74
4.1.4 شرح بسيط لحلقة الأحداث 75
4.1.5 تنفيذ طابور الدوال 76
4.1.6 طابور الرسائل Message Queue 78
4.1.7 طابور العمل Job Queue الخاص بالإصدار ES6 78
4.2 المؤقتات Timers: التنفيذ غير المتزامن في أقرب وقت ممكن 79
4.2.1 الدالة setTimeout() 80
4.2.2 الدالة setImmediate 81
4.2.3 التأخير الصفري Zero delay 81
4.2.4 الدالة setInterval() 82
4.2.5 دالة setTimeout العودية 82
4.3 مطلق الأحداث Event Emitter الخاص بنود Node 84
5. البرمجة غير المتزامنة في Node.js 86
5.1 دوال رد النداء Callbacks 87
5.1.1 معالجة الأخطاء في دوال رد النداء 88

5.1.2 مشكلة دوال رد النداء 88
5.1.3 بدائل دوال رد النداء 89
5.2 الوعود Promises 89
5.2.1 مدخل إلى الوعود 89
5.2.2 إنشاء وعد 90
5.2.3 استهلاك وعد 90
5.2.4 سلسلة الوعود Chaining promises 92
5.2.5 معالجة الأخطاء 93
5.3 صيغة عدم التزامن أو الانتظار async/await 94
5.3.1 كيفية عمل صيغة async/await 94
5.3.2 تطبيق الوعود على كل شيء 95
5.3.3 استخدام دوال متعددة غير متزامنة ضمن سلسلة 96
5.3.4 سهولة تنقيح الأخطاء 97
6. التعامل مع طلبيات الشبكية في Node.js 98
6.1 كيفية عمل بروتوكول HTTP 98
6.1.1 مستندات HTML 99
6.1.2 الروابط والطلبيات 99
6.1.3 توابع HTTP 100
6.1.4 اتصال HTTP خادم/عميل 101
6.1.5 بروتوكول HTTPS والاتصالات الآمنة 103
6.2 كيفية عمل طلبات HTTP 105
6.2.1 تحليل طلبات URL 105
6.2.2 مرحلة بحث DNS 105
6. gethostbyname 105
6.2.3 إنشاء اتصال/مصافحة handshaking طلب TCP 107
6.2.4 إرسال الطلب 107
6.2.5 الاستجابة Response 108
6.2.6 تحليل HTML 108
6.2.7 بناء خادم HTTP باستخدام Node.js 109

6.2.8 إجراء طلبات HTTP 110
ا. إجراء طلب GET 110
ب. إجراء طلب POST 110
ج. PUT و DELETE 111
6.3 مكتبة Axios 111
6.3.1 تثبيت Axios 112
6.3.2 واجهة برمجة تطبيقات Axios 112
6.3.3 إرسال واستقبال الطلبات 113
6.4 مقابس الويب Websockets 115
6.4.1 مقابس الويب الآمنة 116
6.4.2 إنشاء اتصال WebSockets جديد 116
6.4.3 إرسال البيانات إلى الخادم باستخدام WebSockets 116
6.4.4 استقبال البيانات من الخادم باستخدام WebSockets 117
6.4.5 تطبيق خادم WebSockets في Node.js 117
7. التعامل مع الملفات في Node.js 118
7.1 واصفات الملفات File descriptors 118
7.2 إحصائيات الملف 119
7.3 مسارات الملفات 120
7.4 قراءة الملفات 122
7.5 كتابة الملفات 123
7.6 إلحاق محتوى بملف 124
7.7 استخدام المجاري streams 124
7.7.1 مفهوم المجاري streams 124
7.7.2 أنواع المجاري المختلفة 126
7.7.3 كيفية إنشاء مجرى قابل للقراءة 127
7.7.4 كيفية إنشاء مجرى قابل للكتابة 127
7.7.5 كيفية الحصول على بيانات من مجرى قابل للقراءة 127
7.7.6 كيفية إرسال بيانات إلى مجرى قابل للكتابة 128
7.7.7 إعلام مجرى قابل للكتابة بانتهاء الكتابة 128

7.8 التعامل مع المجلدات 129
7.8.1 التحقق من وجود مجلد 129
7.8.2 إنشاء مجلد جديد 129
7.8.3 قراءة محتوى مجلد 129
7.8.4 إعادة تسمية مجلد 130
7.8.5 إزالة مجلد 130
8. تعرف على وحدات Node.js الأساسية 132
8.1 وحدة fs 132
8.2 وحدة المسار path 135
8.2.1 التابع ()path.basename 135
8.2.2 التابع ()path.dirname 136
8.2.3 التابع ()path.extname 136
8.2.4 التابع ()path.isAbsolute 136
8.2.5 التابع ()path.join 136
8.2.6 التابع ()path.normalize 136
8.2.7 التابع ()path.parse 136
8.2.8 التابع ()path.relative 137
8.2.9 التابع ()path.resolve 137
8.3 وحدة os 138
8.3.1 التابع ()os.arch 139
8.3.2 التابع ()os.cpus 139
8.3.3 التابع ()os.endianness 140
8.3.4 التابع ()os.freemem 140
8.3.5 التابع ()os.homedir 140
8.3.6 التابع ()os.hostname 140
8.3.7 التابع ()os.loadavg 140
8.3.8 التابع ()os.networkInterfaces 140
8.3.9 التابع ()os.platform 141
8.3.10 التابع ()os.release 142

8.3.11 التابع ()os.tmpdir 142
8.3.12 التابع ()os.totalmem 142
8.3.13 التابع ()os.type 142
8.3.14 التابع ()os.uptime 142
8.3.15 التابع ()os.userInfo 142
8.4 وحدة الأحداث events 142
8.4.1 التابع ()emitter.addListener 143
8.4.2 التابع ()emitter.emit 143
8.4.3 التابع ()emitter.eventNames 143
8.4.4 التابع ()emitter.getMaxListeners 144
8.4.5 التابع ()emitter.listenerCount 144
8.4.6 التابع ()emitter.listeners 144
8.4.7 التابع ()emitter.off 144
8.4.8 التابع ()emitter.on 144
8.4.9 التابع ()emitter.once 144
8.4.10 التابع ()emitter.prependListener 145
8.4.11 التابع ()emitter.prependOnceListener 145
8.4.12 التابع ()emitter.removeAllListeners 145
8.4.13 التابع ()emitter.removeListener 145
8.4.14 التابع ()emitter.setMaxListeners 145
8.5 وحدة HTTP 145
8.5.1 الخاصيات 146
ا. الخاصية http.METHODS 146
ب. الخاصية http.STATUS_CODES 147
ج. الخاصية http.globalAgent 149
8.5.2 التوابع 149
ا. التابع ()http.createServer 149
ب. التابع ()http.request 149
ج. التابع ()http.get 149

8.5.3 الأصناف Classes 150
ا. الصنف http.Agent 150
ب. الصنف http.ClientRequest 150
ج. الصنف http.Server 150
د. الصنف http.ServerResponse 151
ه. الصنف http.IncomingMessage 151
8.6 وحدة MySQL 152
8.6.1 تثبيت حزمة نود mysql 152
8.6.2 تهيئة الاتصال بقاعدة البيانات 152
8.6.3 خيارات الاتصال 153
8.6.4 إجراء استعلام SELECT 154
8.6.5 إجراء استعلام INSERT 155
8.6.6 إغلاق الاتصال 155
8.7 وحدات مخصصة 156
8.8 الخاتمة 157

دورة تطوير واجهات المستخدم

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

تمهيد

مرحبًا بك في كتاب "دليلك إلى Node.js"، هذا الكتاب بمثابة دليل مبسط لتعلم أساسيات بيئـة Node.js التي تسمح لك بتنفيذ شيفرات لغة جافا سكريبت خارج المتصفح، والتعـرف عليهـا وعلى أساسـياتها وعلى أهم أجزائها شائعة الاستخدام إذ يعتمد على مبدأ 80/20 أي أنه يشـرح أهم 20% من أساسـيات التعامـل مـه البيئـة التي تُستخدم في 80% في الحياة العملية.

هذا الكتاب موجه للمبتدئين فهو يوفر مقدمة شاملة في اسـتخدام Node.js ويركـز على شـرح الأساسـيات والمبادئ التي تحتاجها للبدء في استخدام Node.js بكفاءة بناء تطبيقات الويب باستخدامه إلا أنه لا يتعمـق في شرح المواضيع المتقدمة ويمكنك الاعتماد على مصادر أخرى تغطي مواضيع أكثر تقدمًا.

1. مقدمة إلى Node.js

تُعَدّ Node.js بيئة تشغيل جافا سكريبت JavaScript تعمل من طرف الخادم، وهي مفتوحة المصدر ومتعددة المنصات cross-platform -أي تعمل على أكثر من نظام تشغيل- ومبنية على محرك جافا سكريبت V8 Chrome، وهي تستعمل أساسيًا لإنشاء خوادم الويب، لكنها ليست محدودةً لهذه المهمة فقط، كما أنها لاقت رواجًا بدءًا من انطلاقها في 2009، وتلعب الآن دورًا مهمًا في عالم تطوير الويب، فإذا عددنا أنّ النجوم التي يحصل عليها المشروع في GitHub معيارًا لشهرة البرمجية، فاعلم أنّ Node.js قد حصدت أكثر من 100 ألفًا من النجوم حتى الآن، ومن الجدير بالذكر أنّ Node.js تُستعمل بصورة أساسية لإنشاء خوادم الويب، لكنها ليست محدودةً لهذه المهمة فقط.

1.1 أفضل مميزات Node.js

تتميز Node.js بالمزايا التالية:

1.1.1 السرعة

إحدى الميزات التي تشتهر بها Node.js هي السرعة، فشيفرة JavaScript التي تعمل على Node.js اعتمادًا على اختبارات الأداء benchmark يمكن أن تكون بضعفي سرعة تنفيذ اللغات المصرَّفة compiled مثل C أو Java، وأضعاف سرعة اللغات المفسَّرة مثل بايثون أو روبي بسبب نموذج عدم الحجب non-blocking الذي تستعمله.

1.1.2 البساطة

صدِّقنا عندما نخبرك بأن Node.js بسيطة، بل بسيطة جدًا إذ يمكن للمطورين إنشاء تطبيقات قوية وفعالة باستخدامها بسهولة، كما توفر مجموعة واسعة من المكتبات والأدوات التي تسهل على المطورين عملية بناء التطبيقات بسرعة وكفاءة.

1.1.3 تستعمل JavaScript

تشغِّل Node.js شيفرة جافا سكريبت JavaScript، وهذا يعني أنّ ملايين مبرمجي الواجهات الأمامية الذين يستعملون لغة البرمجة JavaScript في المتصفح سيستطيعون العمل على شيفرات من طرف الخادم ومن طرف الواجهات الأمامية باستخدام اللغة نفسها، فلا حاجة إلى تعلّم لغة جديدة كليًا، حيث ستكون جميع التعابير التي تستعملها في JavaScript متاحةً في Node.js، واطمئن إلى أنّ آخر معايير ECMAScript مستعملة في Node.js، فلا حاجة إلى انتظار المستخدِمين ليحدِّثوا متصفحاتهم، فأنت صاحب القرار بأي نسخة ECMAScript تريد استخدامها في برنامجك باختيار إصدار Node.js المناسب.

1.1.4 تستخدم محرك V8

يمكنك الاستفادة من عمل آلاف المهندسين الذين جعلوا -ويستمروا بجعل- محرك JavaScript الخاص بمتصفح Chrome سريعًا للغاية، وذلك باعتماد Node.js على محرك Chrome V8 المفتوح المصدر.

1.1.5 منصة غير متزامنة

تُعَدّ جميع الأوامر البرمجية في لغات البرمجة التقليدية حاجبة blocking افتراضيًا مثل سي C وجافا Java وبايثون وبي إتش بي PHP؛ أي أن جميع الأوامر البرمجية فيها تنفيذ بالتسلسل، وتوقف التنفيذ حتى اكتمال الأمر الحالي قبل الانتقال إلى الأمر التالي إلا إذا تدخّلتَ بصورة صريحة لإنشاء عمليات غير متزامنة، فإذا أجريت مثلًا طلبًا شبكيًا لقراءة ملف JSON، فسيتوقف التنفيذ حين يكون الرد response جاهزًا.

تسمح جافا سكريبت JavaScript بكتابة شيفرات غير متزامنة asynchronous وغير حاجبة non-blocking بطريقة سهلة جدًا باستخدام خيط thread وحيد ودوال رد النداء callback functions والبرمجة التي تعتمد على الأحداث event-driven، حيث نمرر دالة رد نداء والتي ستُستدعَى حين نتمكن من إكمال معالجة العملية وذلك في كل مرة تحدث عملية تستهلك الموارد، كما أننا لن ننتظر الانتهاء من ذلك قبل الاستمرار في تنفيذ بقية البرنامج.

أُخِذت هذه الآلية من المتصفح، فلا يمكننا انتظار تحميل شيء ما عبر طلب AJAX قبل أن نكون قادرين على التعامل مع أحداث النقر على عناصر الصفحة، فيجب حدوث كل شيء في الوقت الحقيقي لتوفير تجربة جيدة للمستخدِم، مما يسمح بمعالجة آلاف الاتصالات بخادم Node.js وحيد دون الدخول في تعقيدات إدارة الخيوط threads، والتي تكون سببًا رئيسيًا للعلل في البرامج.

> ملاحظة: إذا أنشأتَ المعالج onclick لصفحة الويب، فأنت تستعمل تقنيات البرمجة غير المتزامنة باستخدام معالجات الأحداث في JavaScript.

توفِّر Node.js تعاملًا غير حاجب مع الدخل والخرج I/O، وتكون المكتبات في Node.js عمومًا مكتوبةً بمنهجية عدم الحجب، مما يجعل سلوك الحجب في Node.js استثناءً للقاعدة وليس شيئًا طبيعيًا، كما تكمل Node.js العمليات عند وصول الرد عندما تريد إجراء عملية دخل أو خرج مثل القراءة من الشبكة أو الوصول إلى قاعدة البيانات أو نظام الملفات بدلًا من حجب الخيط blocking the thread وإهدار طاقة المعالج بالانتظار.

1.1.6 عدد هائل من المكتبات

ساعد مدير الحزم npm ببنيته البسيطة النظام العام في node.js، إذ يستضيف npm ما يقرب من 500 ألف حزمة مفتوحة المصدر تستطيع استخدامها بحرّية.

1.2 مثال عن تطبيق Node.js

المثال الأكثر شيوعًا عن تطبيق Node.js هو خادم ويب يعرض العبارة الشهيرة Hello World:

```
const http = require('http')
const hostname = '127.0.0.1'
const port = 3000
const server = http.createServer((req, res) => {
res.statusCode = 200
res.setHeader('Content-Type', 'text/plain')
res.end('Hello World\n')
})
server.listen(port, hostname, () => {
console.log(`Server running at http://${hostname}:${port}/`)
})
```

احفظ الشيفرة البسيطة السابقة في ملف باسم server.js ثم نفِّذ الأمر `node server.js` في الطرفية الخاصة بك وذلك من أجل تنفيذ تلك الشيفرة.

تبدأ الشيفرة السابقة بتضمين وحدة http، إذ تمتلك Node.js مكتبةً قياسيةً رائعةً بما في ذلك دعم التعامل مع الشبكات؛ أما التابع `createServer()` الخاص بوحدة `http` فيُنشئ خادم HTTP جديد ويعيده، كما أنّ الخادم قد ضُبِط للاستماع إلى منفذ وعنوان شبكي محدَّدين، وعندما يجهز الخادم فستُستدعى دالة رد النداء والتي تخبرنا في هذه الحالة أنّ الخادم جاهز، وكلما استقبل الخادم طلبًا request جديدًا، فسيُطلَق الحدث

request الذي يوفّر كائنين هما الطلب أي كائن http.IncomingMessage والرد أي كائن http.ServerResponse، ويُعَدّ هذان الكائنان أساسًا للتعامل مع استدعاء HTTP.

يوفّر الكائن الأول معلومات الطلبية لكننا لم نستعمله في هذا المثال البسيط، إلا أنه يمكنك الوصول إلى ترويسات الطلب وإلى بيانات الطلب؛ أما الكائن الثاني فيعيد البيانات إلى صاحب الطلب caller، وفي هذه الحالة باستخدامنا للسطر التالي:

```
res.statusCode = 200
```

ضبطنا قيمة الخاصية statusCode إلى 200 والتي تعني أنّ الرد ناجح، ثم ضبطنا ترويسة Content-Type كما يلي:

```
res.setHeader('Content-Type', 'text/plain')
```

ثم ننهي الطلب بإضافة المحتوى على أساس وسيط argument إلى التابع `end()`:

```
res.end('Hello World\n')
```

1.3 أدوات Node.js وأطر عملها

تُعَدّ Node.js منصةً منخفضة المستوى، كما توجد آلاف المكتبات المكتوبة باستخدام Node.js لتسهيل الأمور على المطوّرين وجعلها أكثر متعةً وسلاسةً، إذ أصبح عدد كبير من هذه المكتبات شائعًا بين المطوِّرين، وهذه قائمة غير شاملة للمكتبات التي نرى أنها تستحق التعلم:

- Express: أحد أبسط وأقوى الطرق لإنشاء خادم ويب، ويكون تركيزه على البساطة وعدم الانحياز والميزات الأساسية لخادم الويب هو مفتاح نجاحه.
- Meteor: إطار عمل قوي جدًا ومتكامل، يسمح لك ببناء التطبيقات باستخدام JavaScript ومشاركة الشيفرة بين العميل والخادم، كما أصبح الآن يتكامل بسلاسة مع مكتبات واجهة المستخدِم مثل React و Vue و Angular، في حين يمكن استخدامه أيضًا لإنشاء تطبيقات الويب.
- koa: بناه فريق Express نفسه ويطمح إلى أن يكون أصغر وأبسط اعتمادًا على سنوات الخبرة الطويلة للفريق، إذ بدأ هذا المشروع الجديد للحاجة إلى إنشاء تغييرات غير متوافقة مع ما سبقها دون تخريب ما أُنجِز في المشروع.
- Next.js: إطار عمل يستعمل التصيير rendering من طرف الخادم لتطبيقات React.
- Micro: خادم خفيف جدًا لإنشاء خدمات HTTP مصغرة microservices غير متزامنة.
- Socket.io: محرك تواصل في الوقت الحقيقي لبناء تطبيقات الشبكة.

1.4 تاريخ موجز عن Node.js

لننظر إلى تاريخ Node.js من عام 2009 حتى الآن.

أُنشِئت لغة JavaScript في شركة Netscape على أساس أداة لتعديل صفحات الويب داخل متصفحها Netscape Navigator، كما يُعَدّ بيع خوادم الويب جزءًا من نموذج الأعمال في Netscape والتي تحتوي على بيئة باسم Netscape LiveWire التي تستطيع إنشاء صفحات آلية باستخدام JavaScript من طرف الخادم؛ أي أنّ فكرة استخدام JavaScript من طرف الخادم لم تبدأ من Node.js وإنما هي قديمة قدم JavaScript، إلا أنها لم تكن ناجحةً في تلك الفترة.

أحد العوامل الرئيسية لاشتهار Node.js هو التوقيت، إذ بدأت لغة JavaScript تُعَدّ لغةً حقيقيةً، وذلك بفضل تطبيقات Web 2.0 التي أظهرت للعالم كيف تكون التجربة الحديثة للويب مثل Google Maps أو GMail، كما أنّ أداء محركات JavaScript قد ارتفع بشدة بفضل حرب المتصفحات، إذ تعمل فرق التطوير خلف كل متصفح رئيسي بجدّ كل يوم لتحسين الأداء، وكان هذا فوزًا عظيمًا لمنصة JavaScript، علمًا أنّ محرك V8 الذي تستعمله Node.js هو محرك متصفح Chrome، وبالطبع لم تكن شهرة Node.js محض صدفة أو توقيت جيد، إذ أضافت مفاهيم ثورية في كيفية البرمجة باستخدام JavaScript من طرف الخادم.

- عام 2009: ولدت Node.js وأُنشِئت أول نسخة من npm.
- عام 2010: ولد كل من Express و Socket.io.
- عام 2011: وصل npm إلى الإصدار 1.0 وبدأت الشركات الكبيرة مثل LinkedIn بتبني Node.js، كما ولد Hapi.
- عام 2012: استمرت عملية تبنّي Node.js بسرعة كبيرة.
- عام 2013: أُنشِئت أول منصة تدوين باستخدام Node.js: ولدت Koa.
- عام 2014: اشتق مشروع IO.js من Node.js بهدف إضافة دعم ES6 والتطوير بوتيرة أسرع.
- عام 2015: أُسست منظمة Node.js Foundation، ودمج مشروع IO.js مع Node.js مجددًا، كما أصبح npm يدعم الوحدات الخاصة private modules، وأُصدرت نسخة Node 4 (ولم تُصدَر النسخة 1 أو 2 أو 3 من قبل).
- عام 2016: ولد مشروع Yarn، وأُصدِرت Node 6.
- عام 2017: ركّز npm على الحماية أكثر، وصدر Node 8، وأضاف محرك V8 وحدة HTTP/2 بنسخة تجريبية.
- عام 2018: أُصدرت Node 10، وأضيف دعم تجريبي لوحدات ES بلاحقة `mjs.`.

- عام 2019: أُصدرت Node 12 وNode 13.
- عام 2020: أُصدرت Node 14.
- عام 2021: أُصدرت Node 16.
- عام 2022: أُصدرت Node.js 18.
- عام 2023: أُصدرت Node.js 20.
- عام 2024: أُصدرت Node.js 22 (الإصدار المستقر طويل الدعم LTS حاليًا).

وتتضمن كل نسخة أحدث من إصدارات Node.js ميزات وتحسينات جديدة، ويحدَّث معها مدير الحزم npm لتوفير دعم أفضل.

1.5 كيفية تثبيت Node.js

يمكن تثبيت Node.js بطرائق مختلفة، وسنشرح في هذا الفصل أشهر الطرائق وأسهلها لتثبيتها، كما أنّ الحزم الرسمية لجميع أنظمة التشغيل الرئيسية متوافرة على الرابط nodejs.org/en/download.

إحدى الطرائق المناسبة لتثبيت Node.js هي استعمال مدير الحزم، حيث يملك كل نظام تشغيل مدير حزم خاص به، ففي نظام macOS يكون مدير الحزم Homebrew هو مدير الحزم الأساسي، ويسمح بعد تثبيته بتثبيت Node.js بسهولة، وذلك بتنفيذ الأمر التالي في سطر الأوامر داخل الطرفية، ولن نذكر بالتفصيل مدراء الحزم المتاحة للينكس أو ويندوز منعًا للإطالة، لكنها موجودة بالتفصيل في الرابط التالي.

```
brew install node
```

يُعَدّ nvm الطريق الشائع لتشغيل Node.js، إذ يسمح لك بتبديل إصدار Node.js بسهولة وتثبيت الإصدارات الجديدة لتجربتها ثم العودة إلى الإصدار القديم إذا لم يعمل كل شيء على ما يرام، فمن المفيد جدًا على سبيل المثال تجربة الشيفرة الخاصة ببرنامج على الإصدارات القديمة من Node.js، كما ننصحك بمراجعة github.com/creationix/nvm لمزيد من المعلومات حول هذا الخيار، وننصحك بصورة شخصية باستعمال المثبّت الرسمي إذا كنت حديث العهد على Node.js ولا تريد استخدام مدير الحزم الخاص بنظام تشغيلك، لكن على أي حال، سيكون البرنامج التنفيذي node متاحًا في سطر الأوامر بعد تثبيت Node.js بأي طريقة من الطرائق السابقة.

1.6 ماذا عليك معرفته في JavaScript لاستخدام Node.js

إذا بدأت لتوّك مع JavaScript، قد تتساءل ما مدى عمق المعلومات التي تلزمك لاستخدام Node.js؟ من الصعب الوصول إلى النقطة التي تكون فيها واثقًا من قدراتك البرمجية كفاية، فحين تعلّمك كتابة الشيفرات،

فقد تكون محتارًا متى تنتهي شيفرة JavaScript ومتى تبدأ Node.js وبالعكس، لذا ننصحك أن تكون متمكنًا تمكنًا جيدًا من المفاهيم الأساسية في JavaScript قبل التعمق في Node.js:

- بنية البرنامج.
- التعابير.
- أنواع البيانات.
- المتغيرات
- الدوال والدوال السهمية.
- الكلمة المحجوزة this.
- حلقات التكرار والمجالات scopes.
- المصفوفات.
- الفواصل المنقوطة (نعم، هذا المحرف ;)
- الوضع الصارم.
- إصدارات ECMAScript مثل ES6 وES2016 وES2017.

ستكون بعد تعلّمك للمفاهيم السابقة في طريقك لتصبح مطور JavaScript محترف في بيئة المتصفح و Node.js، وفيما يلي مفاهيم أساسية لفهم البرمجة غير المتزامنة asynchronous programming والتي هي جزء أساسي من Node.js:

- مفهوم البرمجة غير المتزامنة ورد النداء callbacks.
- المؤقتات Timers.
- الوعود Promises.
- الكلمتان المحجوزتان Async وAwait.
- التعابير المغلقة Closures.
- حلقات الأحداث.

توجد مقالات كثيرة وكتب في أكاديمية حسوب بالإضافة إلى توثيق JavaScript في موسوعة حسوب عن جميع المواضيع السابقة، ويمكنك بدء التعلم من هذه الصفحة الشاملة التي تلخص لك أبرز مميزات اللغة وأهم مصادر تعلمها.

1.7 الاختلافات بين Node.js والمتصفح

كيف تختلف كتابة تطبيقات JavaScript في Node.js عن البرمجة للويب داخل المتصفح؟ يستخدِم كل من المتصفح و Node.js لغة البرمجة JavaScript؛ لكن بناء التطبيقات التي تعمل في المتصفح مختلف تمامًا عن بناء تطبيقات Node.js، فعلى الرغم من أنهما يستعملان لغة البرمجة نفسها JavaScript، إلا أنه هنالك اختلافات جوهرية تجعل الفرق بينهما كبيرًا.

يملك مطور واجهات المستخدِم الذي يكتب تطبيقات Node.js ميزةً رائعةً ألا وهي استخدامه للغة البرمجة نفسها جافا سكريبت JavaScript، فمن المعروف أنه من الصعب تعلّم لغة برمجة جديدة بإتقان، لكن باستعمال لغة البرمجة نفسها لإجراء كل العمل على موقع الويب سواءً للواجهة الأمامية أو الخادم، حيث توجد أفضلية واضحة في هذه النقطة؛ إلا أنّ بيئة العمل هي التي تختلف.

ستتعامل أغلب الوقت في المتصفح مع شجرة DOM أو غيرها من الواجهات البرمجية الخاصة بالمتصفح Web Platform APIs مثل ملفات تعريف الارتباط Cookies والتي لا توجد في Node.js، وبالطبع لن تكون الكائنات `document` و `window` وغيرها من كائنات المتصفح متوفرةً، كما لن نحصل على الواجهات البرمجية APIs التي توفرها Node.js عبر وحداتها مثل الوصول إلى نظام الملفات.

يوجد اختلاف كبير آخر وهو أنك تستطيع التحكم في البيئة التي تعمل فيها Node.js ما لم تبني تطبيقًا مفتوح المصدر يمكن لأي شخص نشره في أيّ مكان، فأنت تعلم ما هي نسخة Node.js التي سيعمل عليها تطبيقك؛ فليس لديك الحرية في اختيار المتصفح الذي يستخدمه زوار موقعك بموازنة ذلك مع بيئة المتصفح، وهذا يعني أنك يمكنك أن تكتب شيفرات ES6-7-8-9 التي تدعمها نسخة Node.js عندك.

ستكون ملزمًا باستخدام إصدارات JavaScript/ECMAScript القديمة على الرغم من تطوّر JavaScript بسرعة كبيرة، وذلك لأن المتصفحات أبطأ منها والمستخدِمون أبطأ بالتحديث، وصحيحٌ أنك تستطيع استخدام Babel لتحويل شيفرتك إلى نسخة موافقة لمعيار ECMAScript 5 قبل إرسالها إلى زوار موقعك، لكنك لن تحتاج إلى ذلك في Node.js.

> Babel هو أداة تُستخدم في تطوير الويب وبرمجة جافا سكريبت فهو يحول كود جافا سكريبت الحديث إلى نسخة متوافقة مع المتصفحات القديمة التي قد لا تدعم هذه الميزات.

يوجد اختلاف آخر هو استخدام Node.js لنظام الوحدات CommonJS، بينما بدأنا نرى أنّ المتصفحات تستخدم معيار وحدات ES Modules؛ وهذا يعني عمليًا أنه عليك استخدام `require()` في Node.js، و`import` في المتصفح حاليًا.

1.8 محرك V8

V8 هو اسم محرك جافا سكريبت الذي يُشغِّل متصفح Chrome، حيث يأخذ شيفرات جافا سكريبت وينفذها أثناء التصفح عبر Chrome، إذ يوفر بيئة التشغيل اللازمة لتنفيذ شيفرات جافا سكريبت، في حين يوفِّر المتصفح شجرة DOM وغيرها من الواجهات البرمجية للويب، كما يُعَدّ استقلال محرك JavaScript عن المتصفح الذي يستخدمه الميزة الرئيسية التي سببت بانتشار Node.js.

اختار مجتمع مطوري Node.js محرك V8 في 2009، وبعد أن ذاع صيت Node.js صار محرك V8 هو المحرك الذي يشغِّل عددًا غير محصور من شيفرات JavaScript التي تعمل من طرف الخادم، وخذ بالحسبان أنّ بيئة تشغيل Node.js كبيرة جدًا ويعود الفضل إليها في أنّ محرك V8 أصبح يشغِّل تطبيقات سطح المكتب عن طريق مشاريع مثل Electron.js.

1.8.1 محركات جافا سكريبت الأخرى

تملك المتصفحات الأخرى محركات JavaScript مختلفة منها:

- يملك متصفح Firefox محرك SpiderMonkey.
- يملك متصفح Safari محرك JavaScriptCore ويسمى Nitro أيضًا.
- يملك متصفح Edge محرك Chakra.

تطبِّق جميع هذه المحركات معيار ES-262 ECMA والذي يسمى ECMAScript أيضًا، وهو المعيار المستخدَم في لغة JavaScript.

1.8.2 السعي إلى الأداء الأفضل

كُتِب محرك V8 بلغة البرمجة ++C، وهو يُحسَّن باستمرار، ويتميز بكونه محمولاً portable ويعمل على كل من أنظمة ماك ولينكس وويندوز وغيرها من أنظمة التشغيل، كما لن نخوض في تفاصيل محرك V8 لأنها موجودة في مواقع كثيرة مثل موقع V8 الرسمي وتتغير مع مرور الوقت، وعادةً تكون التغييرات كبيرةً، إذ يتطور محرك V8 دومًا كما في غيره في محركات JavaScript لتسريع الويب وبيئة Node.js، كما يجري سباق في عالم الويب للأداء الأفضل منذ سنوات، ونحن -المستخدِمون والمطوِّرون- نستفيد كثيرًا من هذه المنافسة لأنها تعطينا أداءً أسرع وأفضل سنةً بعد سنة.

1.8.3 التصريف Compilation

تُعَدّ لغة JavaScript عمومًا لغةً مفسَّرةً interpreted؛ لكن محركات JavaScript الحديثة لم تَعُد تفسِّر شيفرات JavaScript وحسب، وإنما تُصرِّفها compile، وقد حدث ذلك منذ عام 2009، عندما أُضيف مُصرِّف

SpuderMonkey إلى متصفح Firefox 3.5، ثم اتبع الجميع هذه الفكرة، حيث تُصرَّف JavaScript داخليًا في V8 حين اللزوم بتصريف JIT (اختصارًا لـ just-in-time) لتسريع عملية التنفيذ.

قد ترى أنّ ذلك منافٍ للمنطق، لكن تطورت لغة JavaScript من لغة تنفِّذ عادةً بضعة مئات من الأسطر إلى لغة تشغِّل تطبيقات كاملة تحتوي على آلاف أو حتى مئات الآلاف من الأسطر البرمجية التي تعمل في المتصفح وذلك منذ إطلاق Google Maps عام 2004، فيمكن لتطبيقاتنا الآن أن تعمل لساعات في المتصفح بدلًا من كونها مجرد سكربتات بسيطة للتحقق من صحة المدخلات أو إجراء أفعال بسيطة معينة، ففي هذا العالم الجديد أصبح من المنطقي تمامًا تصريف شيفرات JavaScript؛ وصحيحٌ أنها قد تأخذ بعض الوقت القليل لجهوزية شيفرة JavaScript، إلا أننا سنرى أداءً أفضل من الشيفرة المفسَّرة فقط.

1.9 تشغيل تطبيق Node.js والخروج منه

الطريقة التقليدية لتشغيل برنامج Node.js هي استخدام الأمر `node` المتاح في نظام التشغيل بعد تثبيت Node.js ثم تمرير اسم الملف الذي تريد تنفيذه إلى الأمر، فلو كان تطبيق Node.js عندك موجود في ملف باسم `app.js`، فيمكنك تشغيله بكتابة الأمر التالي في سطر الأوامر:

```
node app.js
```

لنتعلم كيف يمكننا إنهاء تطبيق Node.js بأفضل الطرائق الممكنة، حيث توجد عدة طرق لإنهائه، فعندما تشغِّل برنامج في سطر الأوامر، يمكنك أن توقفه بالضغط على مفتاحي Ctrl+C، لكن ما نريد الحديث عنه هو إنهاء التطبيق برمجيًا، ولنبدأ بأكثر طريقة قاسية لإنهاء التطبيق، ولنرَ لماذا لا يفترض بك استعمالها.

توفِّر الوحدة الأساسية core module المسماة `process` تابعًا يسمح لك بالخروج برمجيًا من تطبيق Node.js وهو `process.exit()`، إذ سيؤدي تشغيل Node.js هذا السطر إلى إغلاق العملية process مباشرةً، وهذا يعني أنه سيتوقف أيّ رد نداء معلَّق، أو أيّ طلب شبكي لا يزال مُرسَلًا، أو أيّ وصول إلى نظام ملفات، أو عمليات تكتب إلى مجرى الخرج القياسي `stdout` أو الخطأ القياسي `stderr` توقفًا مباشرًا دون سابق إنذار، وإذا لم تجد حرجًا في ذلك، فيمكنك تمرير عدد صحيح إلى التابع ليخبر نظام التشغيل ما هي حالة الخروج exit code:

```
process.exit(1)
```

تكون حالة الخروج الافتراضية هي 0، والتي تعني نجاح تنفيذ البرنامج، حيث تمتلك حالات الخروج المختلفة معانٍ خاصة والتي يمكنك استخدامها في نظامك للتواصل مع بقية البرامج، كما يمكنك أيضًا ضبط قيمة الخاصية `process.exitCode` بحيث تُعيد Node.js حالة الخروج المضبوطة إلى هذه الخاصية عند انتهاء تنفيذ البرنامج:

```
process.exitCode = 1
```

ينتهي البرنامج بسلام عند انتهاء تنفيذ جميع الشيفرات فيه في الحالة الطبيعية، وكثيرًا ما نُنشِئ خوادم باستخدام Node.js، على سبيل المثال ينشئ الكود التالي خادم HTTP بسيط:

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
 res.send('Hi!')
})
app.listen(3000, () => console.log('Server ready'))
```

ينشئ خادم ويب بسيط يستجيب بعرض الرسالة 'Hi!' عند الوصول إلى الصفحة الرئيسية. ولن ينتهي هذا البرنامج أبدًا، فإذا استدعيت التابع `process.exit()`، فستنتهي جميع الطلبات قيد التنفيذ أو المعلّقة، وهذا ليس أمرًا جميلًا صدقًا، حيث ستحتاج في هذه الحالة إلى إرسال الإشارة SIGTERM إلى الأمر،

ستتعامل مع الأمر وتنظم إغلاق الخادم بشكل صحيح قبل إنهاء العملية باستخدام معالج إشارة العملية process signal handler كما في الكود التالي:

> لا حاجة إلى تضمين الوحدة `process` باستخدام require، فهي متاحة تلقائيًا.

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
 res.send('Hi!')
})
app.listen(3000, () => console.log('Server ready'))
process.on('SIGTERM', () => {
 app.close(() => {
   console.log('Process terminated')
 })
})
```

نغلق الخادم هنا بعد استلام الإشارة، وقد تتساءل ما هي الإشارات؟ الإشارات هي نظام تواصل داخلي في معيار POSIX، وهو إشعار يُرسَل إلى العملية لإخبارها أنّ حدثًا ما قد حدث، فالإشارة `SIGKILL` هي الإشارة التي تخبر العملية بالتوقف عن العمل فورًا، وهي تعمل مثل `process.exit()`؛ أما الإشارة `SIGTERM` فهي الإشارة التي تخبر العملية بأن تنتهي بلطف، وهي الإشارة التي تُرسَل من مدراء العمليات في أنظمة التشغيل، كما يمكنك إرسال هذه الإشارة من داخل البرنامج باستخدام دالة تابع آخر:

```
process.kill(process.pid, 'SIGTERM')
```

أو من برنامج Node.js آخر، أو أيّ برنامج يعمل على النظام يعرف مُعرِّف PID للعملية المراد إنهاؤها.

1.10 متغيرات البيئة: الفرق بين التطوير والإنتاج

متغير البيئة environment variable هو اصطلاح مُستخدَم على نطاق واسع في المكتبات الخارجية، وسنتعلم كيفية قراءة واستخدام متغيرات البيئة في برنامج Node.js، حيث توفِّر الوحدة الأساسية process في Node.js الخاصية env التي تحتوي على جميع متغيرات البيئة المضبوطة في لحظة تشغيل العملية، وفيما يلي مثال يصل إلى متغير البيئة NODE_ENV المضبوط على القيمة development افتراضيًا:

```
process.env.NODE_ENV // "development"
```

في حال ضبطه بالقيمة production قبل تشغيل السكربت سيخبر Node.js أنّ هذه بيئة إنتاجية وليست تطويرية، كما يمكنك بالطريقة نفسها الوصول إلى أيّ متغيرات بيئة خاصة تضبطها.

يمكن أن يكون لديك إعدادات مختلفة لبيئات الإنتاج والتطوير، حيث يفترض Node أنه يعمل دائمًا في بيئة تطوير، ولكن يمكنك إعلام Node.js بأنك تعمل في بيئة إنتاج من خلال ضبط متغير البيئة NODE_ENV=production عن طريق تنفيذ الأمر التالي:

```
export NODE_ENV=production
```

لكن يُفضَّل في الصدَفة shell وضعه في ملف إعداد الصدَفة مثل .bash_profile مع صدفة Bash، وذلك لأن الإعداد بخلاف ذلك لا يستمر في حالة إعادة تشغيل النظام، كما يمكنك تطبيق متغير البيئة عن طريق وضعه في بداية أمر تهيئة تطبيقك كما يلي:

```
NODE_ENV=production node app.js
```

يضمن ضبط البيئة على القيمة production ما يلي:

- الاحتفاظ بتسجيل الدخول إلى المستوى الأدنى الأساسي.
- إجراء مزيد من مستويات التخبئة أو التخزين المؤقت caching لتحسين الأداء.

تطبِّق مكتبة القوالب Pug التي يستخدمها إطار عمل Express على سبيل المثال عملية التصريف في وضع تنقيح الأخطاء، إذا لم يُضبَط المتغير NODE_ENV على القيمة production، حيث تُصرَّف عروض Express في كل طلب في وضع التطوير، بينما تُخزَّن مؤقتًا في وضع الإنتاج، كما يوفِّر إطار Express خطّافات إعداد configuration hooks خاصة بالبيئة تُستدعَى تلقائيًا بناءً على قيمة المتغير NODE_ENV:

```
app.configure('development', () => {
  //...
})
app.configure('production', () => {
  //...
})
app.configure('production', 'staging', () => {
  //...
})
```

يمكنك استخدام ذلك مثلًا لضبط معالجات أخطاء مختلفة في وضع مختلف كما يلي:

```
app.configure('development', () => {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true
}));
})
app.configure('production', () => {
  app.use(express.errorHandler())
})
```

يعرض هذا الكود في بيئة التطوير معلومات مفصلة حول الأخطاء والاستثناءات في صفحة الاستجابة، في حين يعرض في بيئة الإنتاج رسائل أخطاء أقل تفصيلًا للحفاظ على أمان التطبيق.

1.11 استضافة مشاريع Node.js

يمكن استضافة تطبيقات Node.js في أماكن عديدة اعتمادًا على احتياجاتك، وسنذكر لك قائمةً بالخيارات المتاحة أمامك، وهي قائمة غير شاملة بالخيارات التي يمكنك استخدامها لنشر تطبيقك وجعله متاحًا للعامة، وسنرتِّبها من الأبسط والأقل مزايا إلى الأعقد والأقوى:

- أسهل الخيارات على الإطلاق: نفق محلي local tunnel
- نشر التطبيقات دون أي ضبط
  - Glitch
  - Codepen
- الخيارات عديمة الخوادم Serverless
- المنصة على أساس خدمة PAAS

- ◦ Vercel
- ◦ Nanobox
- ◦ Heroku
- ◦ Microsoft Azure
- ◦ Google Cloud Platform

- خادم خاص افتراضي Virtual Private Server أي VPS
- خادم حقيقي Bare metal

لنشرح المزيد عن كل طريقة من طرق استضافة تطبيقات Node.js وإتاحتها للمستخدمين.

1.11.1 نشر التطبيق في نفق محلي

يمكنك نشر تطبيقك وتُخديم الطلبات من حاسوبك باستخدام نفق محلي local tunnel حتى إذا كان لديك عنوان IP ديناميكي، أو كنت تحت NAT، فهذا الخيار مناسب لإجراء بعض الاختبارات السريعة، أو تجربة المنتج أو مشاركة التطبيق مع مجموعة صغيرة من الأشخاص، وهنالك أداة رائعة لذلك متاحة لجميع المنصات اسمها ngrok، فكل ما عليك فعله لاستعمالها هو كتابة `ngrok PORT`، إذ يشير `PORT` هنا إلى المنفذ الذي تريد نشره على الإنترنت، وستحصل على نطاق من ngrok.io، لكن سيسمح لك الاشتراك المدفوع بالحصول على عنوان URL مخصص إضافةً إلى خيارات حماية إضافة (تذكَّر أنك تفتح جهازك إلى الإنترنت)، وهنالك خدمة أخرى يمكنك استخدامها لنشر تطبيقك بهذه الطريقة وهي github.com/localtunnel/localtunnel.

1.11.2 نشر التطبيقات دون أي ضبط

هناك خيارات متاحة لنشر تطبيقات Node.js دون أيّ ضبط يُذكر، وسنذكر من هذه الخيارات منصة Glitch ومنصة Codepen.

Glitch

تُعَدّ Glitch بيئةً تسمح لك ببناء تطبيقاتك بسرعة كبيرة، ورؤيتها حيةً على النطاق الفرعي الخاص بك على glitch.com، فلا يمكنك حاليًا الحصول على نطاق مخصص وهنالك بعض المحدوديات، لكن ستبقى مع ذلك بيئةً رائعةً؛ فهي تحتوي على كامل ميزات Node.js وCDN ومكان تخزين آمن للمعلومات الحساسة، بالإضافة إلى الاستيراد والتصدير من GitHub، كما أنّ هذه الخدمة موفَّرة من الشركة التي تقف خلف FogBugz وTrello والمشاركين في إنشاء StackOverflow.

Codepen

منصة Codepen رائعة أيضًا، فهي تسمح لك بإنشاء مشروع متعدد الملفات ونشره بنطاق مخصص.

1.11.3 الخيارات عديمة الخوادم Serverless

إحدى الطرائق لنشر تطبيقك وعدم الحاجة إلى خادم لإدارته هي استخدام إحدى الخيارات عديمة الخوادم Serverless وهي منهجية لنشر تطبيقاتك على أساس وظائف functions، وهي ترد على نقطة نهاية شبكية network endpoint، وهذا يسمى أيضًا FAAS أي الوظيفة على أساس خدمة Function As A Service، ومن الخيارات الشائعة جدًا نذكر:

- Serverless Framework
- Standard Library

يوفِّر كلا الخيارين طبقة تجريدية abstraction layer لنشر التطبيقات على حلول مثل AWS Lambda وغيرها من حلول FAAS المبنية على Azure أو Google Cloud.

1.11.4 المنصة على أساس خدمة PAAS

مصطلح PASS هو اختصار لعبارة Platform AS A Service أي المنصة على أساس خدمة ويمكنك استخدامها لجعل عملية نشر التطبيقات أكثر سهولة وفعالية، فهي تحمل عنك عناء التفكير في كثير من الأمور عند نشر تطبيقك، ومن هذه المنصات نذكر:

Zeit Now

يُعَدّ Zeit خيارًا مثيرًا للاهتمام، فعندما تكتب الأمر now في الطرفية، فسيتولى أمر نشر تطبيقك كله؛ وهنالك نسخة مجانية مع محدوديات ونسخة مدفوعة بميزات أكثر، حيث ستنسى أنّ هنالك خادم وكل ما عليك فعله هو نشر التطبيق.

Heroku

يُعَدّ Heroku منصةً رائعةً، وهنالك سلسلة فيديوهات عن كيفية نشر التطبيقات عبر Heroku منها فيديو نشر تطبيق React.js ذو واجهات خلفية Node.js على منصة Heroku.

Microsoft Azure

خدمة Azure توفرها Microsoft Cloud، وإليك مقالة أجنبية تفصيلية تشرح خطوات إنشاء تطبيق Node.js في Azure.

Google Cloud Platform

تُعَدّ منصة Google Cloud خيارًا رائعًا لتنظيم تطبيقاتك، ولديهم توثيق جيد عن Node.js.

Vercel

تعد منصة Vercel منصة سحابية قوية تسهل عليك نشر تطبيقات Node.js وتضمن أداءً عاليًا لها، كما تدعم المنصة نشر الوظائف خفية الخوادم Serverless Functions مما يعني مرونة أكبر لتوسيع تطبيقك، وهي تدعم العديد من أطر ومكتبات Node.js مثل Express.js و Next.js.

1.11.5 خادم خاص افتراضي VPS

ستجد في هذا القسم الخيارات الشائعة التي قد تعرفها من قبل، وهي مرتبة من أكثرها سهولةً للمستخدِم:

- DigitalOcean
- Linode
- Amazon Web Services، ونذكر خصوصًا خدمة Amazon Elastic Beanstalk فهي تسهل بعضًا من تعقيدات AWS.

توفِّر لك هذه الخدمات خادم لينكس فار غ يمكنك العمل عليه، ولن نوصي بدليلٍ محدد لهذه الخدمات؛ وهذه ليست جميع الشركات التي توفر خدمات VPS، لكننا عرضنا لك بعضها هنا على سبيل المثال لا الحصر.

1.11.6 خادم حقيقي

يوجد خيار آخر هو خادم حقيقي bare metal، بحيث تثبت عليه توزيعة لينكس وتصله بالإنترنت أو يمكنك استئجار واحد شهريًا، كما في خدمة Vultr Bare Metal.

دورة الذكاء الاصطناعي

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

2. استخدام الوضع التفاعلي في Node.js

ستتعرّف في هذا الفصل على الوضع التفاعلي REPL في Node.js الذي نستخدمه في الطرفية لتقييم تعبير برمجي مكتوب بلغة javascript، كما ستتعلّم كيفية التعامل مع سطر الأوامر في Node.js من تمرير وسائط منه وإرسال مخرجات إليه وقبول مدخلات منه.

2.1 استخدام الوضع التفاعلي REPL

يسمى الوضع التفاعلي في Node.js باسم REPL اختصارًا إلى Read-Evaluate-Print-Loop، أي اقرأ وقدِّر قيمة التعبير البرمجي ثم اطبع الناتج وكرر العملية، وهو طريقة سريعة لاستكشاف ميزات Node.js.

استعملنا الأمر node سابقًا لتشغيل أحد السكربتات:

```
node script.js
```

أما إذا حذفنا اسم الملف، فسندخل في الوضع التفاعلي REPL:

```
node
```

وعندما تجرب الأمر السابق في الطرفية عندك، فسيحدث ما يلي وسيبقى الأمر في وضع السكون وينتظر منك إدخال تعبير ما:

```
$ node
Welcome to Node.js v14.15.0.
Type ".help" for more information.
>
```

> إذا لم تكن تعرف كيف تفتح الطرفية عندك، فيمكنك البحث في جوجل عن ذلك مع ذكر نظام التشغيل الخاص بك، وانظر مقال "مدخل إلى طرفيّة لينكس".

إذا أردنا أن نكون أكثر دقةً، فإن الوضع التفاعلي ينتظر منّا إدخال شيفرة JavaScript، لذا لنبدأ بسطر بسيط:

```
> console.log('test')
test
undefined
>
```

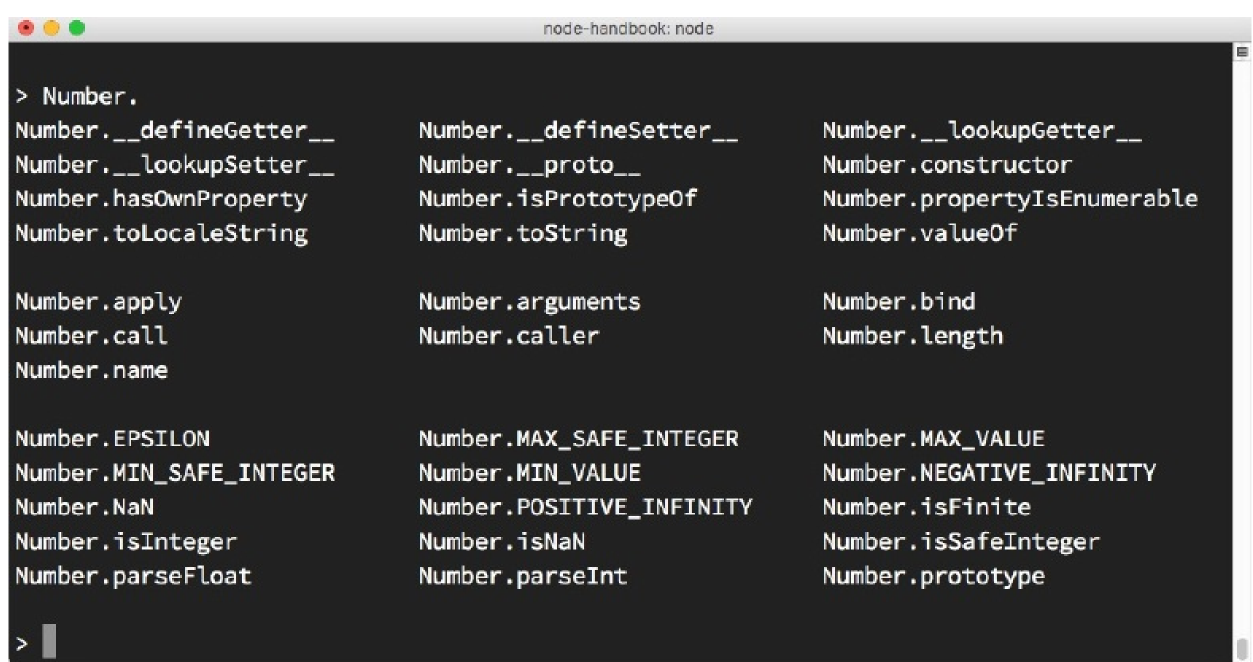
القيمة الأولى المطبوعة `test` هي ناتج الأمر الذي طلبنا منه طباعة سلسلة نصية إلى الطرفية، ثم حصلنا على القيمة `undefined` وهي القيمة المعادة من تنفيذ `console.log()`، ويمكننا الآن إدخال سطر JavaScript جديد.

2.1.1 استخدام الزر tab للإكمال التلقائي

من أجمل مزايا الوضع REPL هو أنه تفاعلي، ففي أثناء كتابك للشيفرة إذا ضغطت على مفتاح tab على لوحة المفاتيح فسيحاول الوضع التفاعلي الإكمال التلقائي لما كتبته ليوافق اسم متغير عرفته مسبقًا.

2.1.2 استكشاف كائنات JavaScript

جرِّب إدخال اسم كائن من كائنات JavaScript مثل `Number` وأضف إليه نقطةً ثم اضغط على tab، إذ سيعرض لك الوضع التفاعلي جميع الخاصيات والتوابع التي يمكنك الوصول إليها في ذاك الكائن.



2.1.3 استكشاف الكائنات العامة

يمكنك معاينة الكائنات العامة بكتابة `global.` ثم الضغط على الزر tab:

```
node-handbook: node
> global.
global.__defineGetter__             global.__defineSetter__
global.__lookupGetter__             global.__lookupSetter__
global.__proto__                    global.constructor
global.hasOwnProperty               global.isPrototypeOf
global.propertyIsEnumerable         global.toLocaleString
global.toString                     global.valueOf

global.Array                        global.ArrayBuffer
global.Boolean                      global.Buffer
global.DTRACE_HTTP_CLIENT_REQUEST   global.DTRACE_HTTP_CLIENT_RESPONSE
global.DTRACE_HTTP_SERVER_REQUEST   global.DTRACE_HTTP_SERVER_RESPONSE
global.DTRACE_NET_SERVER_CONNECTION global.DTRACE_NET_STREAM_END
global.DataView                     global.Date
global.Error                        global.EvalError
global.Float32Array                 global.Float64Array
global.Function                     global.GLOBAL
global.Infinity                     global.Int16Array
global.Int32Array                   global.Int8Array
global.Intl                         global.JSON
global.Map                          global.Math
global.NaN                          global.Number
global.Object                       global.Promise
global.Proxy                        global.RangeError
global.ReferenceError               global.Reflect
global.RegExp                       global.Set
global.String                       global.Symbol
global.SyntaxError                  global.TypeError
global.URIError                     global.Uint16Array
global.Uint32Array                  global.Uint8Array
global.Uint8ClampedArray            global.WeakMap
global.WeakSet                      global.WebAssembly
```

2.1.4 المتغير الخاص _

إذا كتبت الرمز _ بعد شيفرة ما، فسيؤدي ذلك إلى طباعة ناتج آخر عملية.

2.1.5 الأوامر ذوات النقط

يملك الوضع التفاعلي REPL بعض الأوامر الخاصة وكلها تبدأ بنقطة `.` وهي:

- `.help`: يظهر المساعدة للأوامر ذوات النقط.
- `.editor`: تفعيل وضع المحرر، وذلك لكتابة شيفرة JavaScript متعددة الأسطر بسهولة، وبعد دخولك في هذا الوضع وكتابتك للشيفرة المطلوبة، فيمكنك إدخال Ctrl+D لتنفيذ الأمر الذي كتبت.

- `.break`: إذا كتبت الأمر `.break` عند كتابتك لتعبير متعدد الأسطر، فستلغي أي مدخلات قادمة، ومثله مثل الضغط على Ctrl+C.
- `.clear`: إعادة ضبط سياق الوضع التفاعلي إلى كائن فارغ، وإزالة أي تعابير متعددة الأسطر جرت كتابتها.
- `.load`: تحميل ملف JavaScript بمسار نسبي إلى مجلد العمل الحالي.
- `.save`: حفظ ما أدخلته في الجلسة التفاعلية إلى ملف مع تمرير مسار الملف بعد كتابة `.save`.
- `.exit`: الخروج من الوضع التفاعلي وهو يماثل الضغط على Ctrl+C مرتين.

يعرِف الوضع التفاعلي REPL متى تكتب تعبيرًا متعدد الأسطر دون الحاجة إلى تشغيل الأمر `.editor`، فإذا بدأت مثلًا بكتابة حلقة تكرار مثل هذه:

```
[1, 2, 3].forEach(num => {
```

ثم ضغطت على زر `enter`، فسيعرف الوضع التفاعلي أنك تكتب تعبيرًا متعدد الأسطر ويبدأ سطرًا جديدًا بثلاث نقط ...، مما يشير إلى أنك ما زلت تعمل على القسم أو المقطع نفسه:

```
... console.log(num)
... })
```

إذا كتبت `.break` في آخر السطر، فسيتوقف وضع تعدد الأسطر ولن يُنفَّذ التعبير الذي كتبته.

2.2 تمرير الوسائط من سطر الأوامر إلى Node.js

سنشرح في هذا القسم كيفية استقبال وسائط arguments في برنامج Node.js مُمرَّرة من سطر الأوامر، إذ يمكنك تمرير أي عدد من الوسائط أثناء تشغيل برنامج Node.js باستخدام الأمر:

```
node app.js
```

يمكن أن تكون الوسائط بمفردها، أو على شكل زوج من المفاتيح والقيم مثل:

```
node app.js ahmed
```

أو

```
node app.js name=ahmed
```

يجعل هذا طريقة الحصول على القيمة مختلفةً في شيفرة Node.js. إذ أنّ طريقة الحصول على الوسائط هي استخدام الكائن `process` المبني في Node.js، ففيه الخاصية `argv` التي هي مصفوفة تحتوي على

جميع الوسائط الممررة عبر سطر الأوامر؛ ويكون الوسيط الأول هو المسار الكامل للأمر node، بينما الوسيط الثاني هو مسار الملف المُنفَّذ كاملًا، وجميع الوسائط الإضافية ستكون موجودةً من الموضع الثالث إلى آخره، كما يمكنك المرور على جميع المعاملات بما في ذلك مسار node ومسار الملف المُنفَّذ باستخدام حلقة تكرار:

```
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`)
})
```

يمكنك الحصول على الوسائط الإضافية من خلال إنشاء مصفوفة جديدة تستثني أول قيمتين:

```
const args = process.argv.slice(2)
```

إذا كان لديك معامل بمفرده دون مفتاح مثل:

```
node app.js ahmed
```

فيمكنك الوصول إليه كما يلي:

```
const args = process.argv.slice(2)
args[0]
```

أما في حالة كان الوسيط هو مفتاح وقيمة كما في:

```
node app.js name=ahmed
```

فإن قيمة `args[0]` هي `name=ahmed`، وستحتاج إلى تفسيرها، وأفضل طريقة هي استخدام المكتبة `minimist` التي تساعدنا بالتعامل مع الوسائط:

```
const args = require('minimist')(process.argv.slice(2))
args['name'] // ahmed
```

2.3 إرسال تطبيق Node.js المخرجات إلى سطر الأوامر

سنتعلم كيفية الطباعة إلى سطر الأوامر باستخدام Node.js بدءًا من الاستخدام الأساسي للتابع `console.log` حتى وصولنا إلى الأمور المعقدة.

2.3.1 طباعة المخرجات باستخدام الوحدة console

توفِّر Node.js الوحدة `console` التي توفر عددًا كبيرًا من الطرائق المفيدة في التعامل مع سطر الأوامر، وهي تشبه إلى حد ما الكائن `console` الموجود في المتصفحات، والتابع الأساسي الأكثر استخدامًا في هذه الوحدة هو التابع `console.log()`، الذي يطبع السلسلة النصية التي تمررها إليه إلى الطرفية، وإذا مررت كائنًا

فسيعرضه على أساس سلسلة نصية، كما يمكنك تمرير قيم متعددة إلى التـابع `console.log()` كمـا يلي، إذ ستطبع Node.js القيمتين x و y معًا:

```
const x = 'x'
const y = 'y'
console.log(x, y)
```

يمكنك أيضًا تنسيق السلاسل النصية بتمرير القيم ومُحدِّد التنسيق كما في المثال التالي:

```
console.log('My %s has %d years', 'cat', 2)
```

إذ أنّ محددات التنسيق format specifiers هي:

- %s: تنسيق المتغير على أساس سلسلة نصية.
- %d أو `%i`: تنسيق المتغير على أساس عدد صحيح.
- %f: تنسيق المتغير على أساس عدد ذي فاصلة عشرية.
- %O: تنسيق المتغير على أساس كائن كما في المثال البسيط الآتي:

```
console.log('%O', Number)
```

2.3.2 مسح محتوى الطرفية

يمسح التابع `console.clear()` محتوى الطرفية، لكن يختلف سلوكه اعتمادًا على الطرفية المستخدَمة.

2.3.3 عد العناصر

يُعَدّ التابع `console.count()` مفيدًا في بعض الحالات التي تريـد فيهـا عـدّ المـرات الـتي طُبِعَت فيهـا السلسلة النصية وإظهار الرقم بجوارها، خذ هذا المثال:

```
const x = 1
const y = 2
const z = 3
console.count(
  'The value of x is ' + x + ' and has been checked .. how many
times?'
)
console.count(
  'The value of x is ' + x + ' and has been checked .. how many
times?'
```

```
)
console.count(
  'The value of y is ' + z + ' and has been checked .. how many times?'
)
```

يمكننا إحصاء عدد البرتقالات والتفاحات التي لدينا:

```
const oranges = ['orange', 'orange']
const apples = ['just one apple']
oranges.forEach(fruit => {
  console.count(fruit)
})
apples.forEach(fruit => {
  console.count(fruit)
})
```

2.3.4 طباعة تتبع مكدس الاستدعاء

قد تكون هنالك حالات من المفيد فيها طباعة تتبع مكدس الاستدعاء call stack trace لإحدى الدوال، وذلك للإجابة مثلًا على السؤال التالي: كيف وصلنا إلى هذا الجزء من الشيفرة، إذ يمكنك استخدام التابع `console.trace()`:

```
const function2 = () => console.trace()
const function1 = () => function2()
function1()
```

سيطبع مكدس الاستدعاء ما يلي، وهذا هو الناتج إذا جربنا الشيفرة السابقة في النمط التفاعلي REPL في Node.js:

```
Trace
    at function2 (repl:1:33)
    at function1 (repl:1:25)
    at repl:1:1
    at ContextifyScript.Script.runInThisContext (vm.js:44:33)
    at REPLServer.defaultEval (repl.js:239:29)
    at bound (domain.js:301:14)
    at REPLServer.runBound [as eval] (domain.js:314:12)
```

```
    at REPLServer.onLine (repl.js:440:10)
    at emitOne (events.js:120:20)
    at REPLServer.emit (events.js:210:7)
```

2.3.5 طباعة الزمن المستغرق

يمكنك ببساطة حساب مقدار الوقت الذي أخذته الدالة للتنفيذ من خلال استخدام التابعين `time()` و `timeEnd()` كما في المثال التالي:

```
const doSomething = () => console.log('test')
const measureDoingSomething = () => {
  console.time('doSomething()')
  // افعل شيئًا وقس الوقت المستغرق لتنفيذه
  doSomething()
  console.timeEnd('doSomething()')
}
measureDoingSomething()
```

2.3.6 مجرى الخرج القياسي stdout والخطأ القياسي stderr

رأينا أنّ التابع `console.log()` رائع لطباعة الرسائل في الطرفية، وهذا ما يسمى مجرى الخرج القياسي standard output stream أو stdout؛ أما التابع `console.error()` فسيطبع الرسائل المرسَلة إلى مجرى الخطأ القياسي stderr والتي لن تظهر في الطرفية وإنما ستظهر في سجل الخطأ.

2.3.7 طباعة المخرجات بألوان مختلفة

يمكنك أن تغيير لون المخرجات النصية في الطرفية باستخدام تسلسلات التهريب escape sequences، وتسلسل التهريب هو مجموعة من المحارف التي تُعرِّف لونًا معينًا في الطرفية مثل:

```
console.log('\x1b[33m%s\x1b[0m', 'hi!')
```

يمكنك تجربة ذلك باستخدام النمط التفاعلي في Node.js وستُعرَض السلسلة النصية `hi!` باللون الأصفر، لكن هذه الطريقة ذات مستوى منخفض، فأبسط طريقة لتلوين المخرجات في الطرفية هي باستخدام مكتبة مثل Chalk، حيث يمكنها أيضًا إجراء عمليات التنسيق الأخرى إضافةً إلى تلوينها للنص مثل جعل النص غامقًا أو مائلًا أو مسطرًا تحته، كما يمكنك تثبيتها باستخدام `npm install chalk` ثم استخدامها:

```
const chalk = require('chalk')
console.log(chalk.yellow('hi!'))
```

من المؤكد أن استخدام `chalk.yellow` أكثر راحةً من محاولة تذكُّر شيفرات التهريب وستكون قابلية قراءة الشيفرة أفضل.

2.3.8 إنشاء شريط تقدم في الطرفية

تُعَدّ حزمة Progress حزمةً رائعةً لإنشاء شريط تقدُّم في الطرفية، حيث يمكنك تثبيتها باستخدام `npm install progress` ببساطة.

تُنشِئ الشيفرة التالية شريط تقدُّم بعشر خطوات، حيث ستكتمل خطوة كل 100 ميللي ثانية، وحين اكتمال الشريط سنصفِّر العداد:

```
const ProgressBar = require('progress')
const bar = new ProgressBar(':bar', { total: 10 })
const timer = setInterval(() => {
  bar.tick()
  if (bar.complete) {
    clearInterval(timer)
  }
}, 100)
```

2.4 قبول المدخلات من سطر الأوامر

يمكننا جعل تطبيق Node.js الذي يعمل من سطر الأوامر تفاعليًا باستخدام وحدة `readline` المبنية في أساس Node.js.

فقد أصبحت Node.js بدءًا من الإصدار السابع توفِّر الوحدة `readline` التي تحصل على المدخلات من مجرى قابل للقراءة مثل `process.stdin` (مجرى الدخل القياسي stdin) سطرًا بسطر والذي يكون أثناء تنفيذ برنامج Node.js من سطر الأوامر هو الدخل المكتوب في الطرفية.

```
const readline = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout
})
readline.question(`What's your name?`, (name) => {
  console.log(`Hi ${name}!`)
  readline.close()
})
```

تسأل الشيفرة السابقة المستخدِم عن اسمه، وبعد كتابة المستخدِم الاسم والضغط على مفتاح Enter سنرسل تحيةً له.

يُظهِر التابع `question()` المعامل parameter الأول -أي السؤال- وينتظر مدخلات المستخدِم، ثم يستدعي دالة رد نداء عندما يضغط المستخدِم على مفتاح Enter. لاحظ أننا أغلقنا واجهة `readline` في دالة رد النداء، كما توفِّر الواجهة `readline` توابعًا كثيرةً متعددةً وسنتركك لاستكشافها من توثيقها.

أما إذا احتجت إلى إدخال كلمة مرور، فمن الأفضل إظهار رمز `*` بدلًا منها، وأسهل طريقة لذلك هو استخدام الحزمة readline-sync التي تشبه `readline` في الواجهة البرمجية وتستطيع التعامل مع هذه الحالة دون عناء، والخيار الممتاز والمتكامل هو الحزمة inquirer.js، حيث يمكنك تثبيتها باستخدام الأمر `npm install inquirer`، كما يمكنك إعادة كتابة الشيفرة السابقة كما يلي:

```
const inquirer = require('inquirer')
var questions = [{
  type: 'input',
  name: 'name',
  message: "What's your name?",
}]
inquirer.prompt(questions).then(answers => {
  console.log(`Hi ${answers['name']}!`)
})
```

تسمح لك مكتبة inquirer.js بفعل أمور كثيرة مثل الأسئلة متعددة الخيارات وأزرار الانتقاء والتأكيدات وغير ذلك، كما من المفيد معرفة جميع الخيارات المتاحة أمامنا خصوصًا التي توفِّرها Node.js، لكن إذا أردت التعامل مع مدخلات سطر الأوامر كثيرًا، فالخيار الأمثل هو مكتبة inquirer.js.

دورة تطوير التطبيقات باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

3. مدير الحزم npm في Node.js

يمثِّل مدير حزم نود npm -اختصارًا إلى Node Package Manager- أساس نجاح Node.js، فقد صدر تقرير في شهر 1 من عام 2017 بوجود أكثر من 350000 حزمة مُدرجَة في سجل npm، مما يجعله أكبر مستودع لشيفرات لغة على الأرض، فكُن على ثقة أنك ستجد فيه حزمةً لكل شيء تقريبًا.

يُعَدّ npm مدير حزم Node.js المعياري، فقد اُستخدِم في البداية على أساس طريقة لتنزيل وإدارة اعتماديات حزم Node.js، لكنه أصبح بعدها أداةً تُستخدَم في واجهة جافا سكريبت الأمامية أيضًا.

> تُعدّ Yarn أداة إدارة حزم فعالة أيضًا، ويمكن استخدامها كبديل لمدير الحزم npm.

3.1 إدارة تنزيل الحزم والمكتبات والاعتماديات

يدير npm تنزيلات جميع اعتماديات مشروعك.

3.1.1 تثبيت جميع الاعتماديات Dependencies

يمكنك استخدام الأمر التالي إذا احتوى المشروع على ملف packages.json:

```
npm install
```

سيثبِّت هذا الأمر كل ما يحتاجه المشروع في المجلد `node_modules` وينشئ المجلد إن لم يكن موجودًا.

3.1.2 تثبيت حزمة واحدة

يمكنك أيضًا تثبيت حزمة معينة عن طريق تشغيل الأمر:

```
npm install <package-name>
```

سترى في أغلب الأحيان مزيدًا من الرايات flags المضافة إلى هذا الأمر مثل:

- `--save` التي تثبّت وتضيف مدخلة إلى اعتماديات ملف package.json وهي الافتراضية فلا داعي لإضافتها في كل مرة تثبت فيها حزمة في مشروعك.
- `--save-dev` التي تثبّت وتضيف مدخلة إلى اعتماديات تطوير devDependencies ملف `package.json`.

يتمثل الاختلاف الأساسي بينهما في أن اعتماديات التطوير devDependencis هي أدوات تطوير مثل مكتبة الاختبار، بينما تُجمَّع الاعتماديات dependencies مع التطبيق الذي يكون قيد الإنتاج.

يمكن تثبيت إصدار أقدم من حزمة npm أو تثبيت إصدار محدد بعينه، وهو شيء قد يكون مفيدًا في حل مشكلة التوافق، كما يمكنك تثبيت إصدار قديم من حزمة npm باستخدام صيغة `@` كما يلي:

```
npm install <package>@<version>
```

يثبّت الأمر التالي الإصدار الأخير الأحدث من حزمة `cowsay`:

```
npm install cowsay
```

يمكنك تثبيت الإصدار `1.2.0` من خلال الأمر التالي:

```
npm install cowsay@1.2.0
```

يمكن تطبيق الشيء نفسه مع الحزم العامة كما يلي:

```
npm install -g webpack@4.16.4
```

ويمكنك سرد جميع إصدارات الحزمة السابقة باستخدام الأمر `npm view <package> versions` كما يلي:

```
npm view cowsay versions
[  '1.0.0',
  '1.0.1',
  '1.0.2',
  '1.0.3',
  '1.1.0',
  '1.1.1',
  '1.1.2',
  '1.1.3',
  '1.1.4',
  '1.1.5',
  '1.1.6',
  '1.1.7',
  '1.1.8',
  '1.1.9',
  '1.2.0',
  '1.2.1',
  '1.3.0',
  '1.3.1' ]
```

3.1.3 مكان تثبيت npm للحزم

يمكنك إجراء نوعين من التثبيت، عند تثبيت حزمة باستخدام `npm` أو `yarn`:

- تثبيت محلي local install.
- تثبيت عام global install.

إذا كتبتَ أمر تثبيت `npm install` مثل الأمر التالي، فستُثبَّت الحزمة في شجرة الملفات الحالية ضمن المجلد الفرعي `node_modules` افتراضيًا، ويضيف عندها `npm` أيضًا المدخلة `lodash` في خاصية الاعتماديات dependencies الخاصة بملف package.json الموجود في المجلد الحالي:

```
npm install lodash
```

يُطبَّق التثبيت العام باستخدام الراية `-g`:

```
npm install -g lodash
```

لن يثبّت `npm` الحزمة ضمن المجلد المحلي وإنما سيستخدم موقعًا عامًا، إذ سيخبرك الأمر `npm root -g` بمكان هذا الموقع الدقيق على جهازك، حيث يمكن أن يكون هذا الموقع في نظام التشغيل macOS أو لينكس `/usr/local/lib/node_modules/`، ويمكن أن يكون على نظام التشغيل ويندوز `C:\Users\YOU\AppData\Roaming\npm\node_modules`.

لكن إذا استخدمت `nvm` لإدارة إصدارات Node.js، فقد يختلف هذا الموقع، حيث استخدمنا `nvm` وكان موقع الحزم `/Users/flavio/.nvm/versions/node/v8.9.0/lib/node_modules`.

3.1.4 كيفية استخدام أو تنفيذ حزمة مثبتة باستخدام npm

هل تساءلت عن كيفية تضمين واستخدام حزمة مثبَّتة في مجلد `node_modules` في شيفرتك الخاصة؟ حسنًا، لنفترض أنك ثبّت مكتبة أدوات جافا سكريبت الشائعة lodash باستخدام الأمر التالي:

```
npm install lodash
```

سيؤدي ذلك إلى تثبيت الحزمة في مجلد `node_modules` المحلي التي يمكنك استخدامها في شيفرتك الخاصة من خلال استيرادها في برنامجك باستخدام `require`:

```
const _ = require('lodash')
```

إذا كانت حزمتك الخاصة قابلة للتنفيذ، فسيوضَع الملف القابل للتنفيذ ضمن المجلد `node_modules/.bin/`، وإحدى طرق إثبات ذلك هي استخدام الحزمة cowsay، حيث توفّر هذه الحزمة برنامج سطر أوامر يمكن تنفيذه لإنشاء بقرة تقول شيئًا -وحيوانات أخرى أيضًا-، حيث ستثبّت هذه الحزمة نفسها وعددًا من الاعتماديات في المجلد `node_modules` عند تثبيتها باستخدام الأمر `npm install cowsay`:

```
~/dev/node/cowsay/node_modules
❯ ll
drwxr-xr-x - flavio  3 Aug 17:06 ansi-regex
drwxr-xr-x - flavio  3 Aug 17:06 cowsay
drwxr-xr-x - flavio  3 Aug 17:06 get-stdin
drwxr-xr-x - flavio  3 Aug 17:06 is-fullwidth-code-point
drwxr-xr-x - flavio  3 Aug 17:06 minimist
drwxr-xr-x - flavio  3 Aug 17:06 optimist
drwxr-xr-x - flavio  3 Aug 17:06 string-width
drwxr-xr-x - flavio  3 Aug 17:06 strip-ansi
drwxr-xr-x - flavio  3 Aug 17:06 strip-eof
drwxr-xr-x - flavio  3 Aug 17:06 wordwrap

~/dev/node/cowsay/node_modules
❯
```

يوجد مجلد `.bin` مخفي يحتوي على روابط رمزية إلى ملفات cowsay الثنائية:

```
~/dev/node/cowsay/node_modules/.bin
❯ ll
lrwxr-xr-x 16 flavio  3 Aug 17:06 cowsay -> ../cowsay/cli.js
lrwxr-xr-x 16 flavio  3 Aug 17:06 cowthink -> ../cowsay/cli.js

~/dev/node/cowsay/node_modules/.bin
❯
```

يمكنك تنفيذ هذه الحزمة من خلال كتابة `./node_modules/.bin/cowsay` لتشغيلها، لكن يُعَدّ npx المُضمَّن في الإصدارات الأخيرة من npm -منذ الإصدار 5.2- الخيار الأفضل، فما عليك إلا تشغيل الأمر التالي:

```
npx cowsay
```

وسيجد npx موقع الحزمة.

3.1.5 تحديث الحزم

أصبح التحديث سهلًا أيضًا عن طريق تشغيل الأمر:

```
npm update
```

سيتحقّق npm من جميع الحزم بحثًا عن إصدار أحدث يلبي قيود إدارة الإصدارات Versioning الخاصة بك، كما يمكنك تحديد حزمة واحدة لتحديثها أيضًا باستخدام الأمر التالي:

```
npm update <package-name>
```

لتحديث جميع اعتماديات npm المخزَّنة في الملف package.json لأحدث إصدار متاح لها، ثبَّتَ الحزمة باستخدام الأمر `npm install <packagename>` الذي سينزِّل أحدث إصدار متاح من الحزمة ويضعه في مجلد `node_modules`، وتُضاف مدخلة مقابلة إلى الملف `package.json` والملف `package-lock.json` في مجلدك الحالي، إذ يحسب npm الاعتماديات ويثبّت أحدث إصدار متاح منها.

لنفترض أنك ثبَّتَ الحزمة cowsay، وهي أداة سطر أوامر رائعة تتفاعل مع المطور بطريقة لطيفة وتتيح إنشاء بقرة تقول أشياء ضمن سطر الأوامر، فإذا ثبَّتَها بالأمر `npm install cowsay` فسوف يضاف التالي لملف `package.json`:

```
{
  "dependencies": {
    "cowsay": "^1.3.1"
  }
}
```

يمثّل ما يلي جزءًا من ملف package-lock.json، حيث أزلنا الاعتماديات المتداخلة للتوضيح:

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "cowsay": {
      "version": "1.3.1",
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-
1.3.1.tgz",
      "integrity":
"sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BTTDkMAjufp+0F9eLjzRnOH
zVAYeIYFF5po5NjRrgefnRMQ==",
      "requires": {
        "get-stdin": "^5.0.1",
        "optimist": "~0.6.1",
        "string-width": "~2.1.1",
        "strip-eof": "^1.0.0"
      }
    }
  }
```

```
}
```

يوضّح هذان الملفان أننا ثبّتنا الإصدار `1.3.1` من الحزمة cowsay باستخدام قاعـدة التحـديثات `^1.3.1`، والتي تعني بالنسبة لقواعد إدارة إصـدارات npm أنـه يمكن تحـديث npm إلى إصـدار حزمـة التصـحيح patch والإصدار الثانوي minor، أي `0.13.1` و`0.14.0` وما إلى ذلك، فإذا كـان هنـاك إصـدار ثـانوي أو إصـدار حزمـة تصحيح جديد وكتبنا الأمر `npm update`، فسيُحدَّث الإصدار المثبَّت، وسيُملأ ملف `package-lock.json` بالإصدار الجديد، بينما يبقى الملف `package.json` دون تغيير، كما يمكنك اكتشاف إصدارات الحزم الجديـدة من خلال تشغيل الأمر `npm outdated`، وفيما يلي قائمة ببعض الحزم القديمة في مستودع واحـد لم نحـدِّثها لفترة طويلة:

```
~/www/flaviocopes.com/themes/ghostwriter
~/www/flaviocopes.com/themes/ghostwriter master* 1d 31m 26s
❯ npm outdated
Package                             Current   Wanted   Latest  Location
autoprefixer                          8.6.5    8.6.5    9.1.0  ghostwriter
css-loader                           0.28.4  0.28.11    1.0.0  ghostwriter
cssnano                              3.10.0   3.10.0    4.0.5  ghostwriter
extract-text-webpack-plugin           2.1.2    2.1.2    3.0.2  ghostwriter
node-sass                             4.5.3    4.9.2    4.9.2  ghostwriter
normalize.css                         7.0.0    7.0.0    8.0.0  ghostwriter
optimize-css-assets-webpack-plugin    2.0.0    2.0.0    5.0.0  ghostwriter
postcss-cli                           5.0.1    5.0.1    6.0.0  ghostwriter
postcss-discard-comments              2.0.4    2.0.4    4.0.0  ghostwriter
sass-loader                           6.0.6    6.0.7    7.1.0  ghostwriter
style-loader                         0.18.2   0.18.2   0.21.0  ghostwriter
webpack                               3.0.0   3.12.0   4.16.4  ghostwriter

~/www/flaviocopes.com/themes/ghostwriter master*
❯
```

تُعَدّ بعض هـذه التحـديثات إصـدارات رئيسـية، إذ لن يـؤدي تشـغيل الأمـر `npm update` إلى تحـديثها، فالإصدارات الرئيسية لا تُحدَّث بهذه الطريقة أبدًا لأنها حسـب التعريـف تقـدِّم تغيـيرات جذريـة، ولأن npm يريـد توفير المتاعب عليك، في حين يمكنك تحديث جميع الحزم إلى إصـدار رئيسـي جديـد من خلال تثـبيت الحزمـة تثبيتًا عامًا باستخدام الأمر `npm-check-updates` كما يلي:

```
npm install -g npm-check-updates
```

ثم تشغيلها باستخدام الأمر التالي:

```
ncu -u
```

يؤدي ذلك إلى ترقية جميع تلميحات الإصدار في ملف package.json إلى الاعتماديـات dependencies و devDependencies، لذا يستطيع npm تثبيت الإصدار الرئيسي الجديد، ويمكن الآن تشغيل أمر التحديث:

```
npm update
```

إذا حمّلتَ المشروع دون اعتماديات `node_modules` وأردت تثبيت الإصدارات الجديدة أولًا، شغل الأمر:

```
npm install
```

3.1.6 إدارة الإصدارات وسرد إصدارات الحزم المثبتة

يـدير npm أيضًـا -بالإضـافة إلى التـنزيلات العاديـة- عمليـة إدارة الإصـدارات versioning، بحيث يمكنـك تحديد إصدار معيّن من الحزمة، أو طلب إصدار أحـدث أو أقـدم ممـا تحتاجـه، وسـتجد في كثـير من الأحيـان أنّ المكتبة متوافقة فقط مع إصدار رئيسي لمكتبة أخرى، أو قد تجد خطأً غـير مُصـحَّح بعـد في الإصـدار الأخـير من مكتبة، مما يسبّب مشكلات، كما يساعد تحديد إصدار صريح من مكتبـة أيضًـا في إبقـاء كـل فريـق العمـل على إصدار الحزمة الدقيق نفسه، بحيث يشغّل الفريق بأكمله الإصدار نفسه حتى تحديث ملف package.json.

تساعد عملية تحديد الإصـدار كثـيرًا في جميـع الحـالات السـابقة، حيث يتبـع `npm` معيـار إدارة الإصـدارات الدلالية semantic versioning - أو semver اختصارًا- والذي سنشرحه تاليًا في قسـم منفصـل، وقـد تحتـاج عمومًا إلى معرفة إصدار حزمة معينة ثبّتها في تطبيقـك، وهنـا يمكنـك اسـتخدم الأمـر التـالي لمعرفـة الإصـدار الأحدث من جميع الحزم المثبّتة بالإضافة إلى اعتمادياتها:

```
npm list
```

إليك المثال التالي:

```
npm list
/Users/flavio/dev/node/cowsay
└─┬ cowsay@1.3.1
  ├── get-stdin@5.0.1
  ├─┬ optimist@0.6.1
  │ ├── minimist@0.0.10
  │ └── wordwrap@0.0.3
  ├─┬ string-width@2.1.1
  │ ├── is-fullwidth-code-point@2.0.0
  │ └─┬ strip-ansi@4.0.0
  │   └── ansi-regex@3.0.0
  └── strip-eof@1.0.0
```

يمكنك فتح ملف `package-lock.json` فقط، ولكنه يحتاج بعض الفحص البصري، حيث يطبق الأمر `npm list -g` الشيء نفسه ولكن للحزم المثبَّتة تثبيتًا عامًا، كما يمكنك الحصول على حزم المستوى الأعلى فقط، أي الحزم التي طلبتَ من npm تثبيتها وأدرجتَها في ملف `package.json`، من خلال تشغيل الأمر `npm list --depth=0` كما يلي:

```
npm list --depth=0
/Users/flavio/dev/node/cowsay
└── cowsay@1.3.1
```

يمكنك الحصول على إصدار حزمة معينة عن طريق تحديد اسمها كما يلي:

```
npm list cowsay
/Users/flavio/dev/node/cowsay
└── cowsay@1.3.1
```

وتعمل هذه الطريقة أيضًا مع اعتماديات الحزم التي ثبَّتها كما يلي:

```
npm list minimist
/Users/flavio/dev/node/cowsay
└─┬ cowsay@1.3.1
  └─┬ optimist@0.6.1
    └── minimist@0.0.10
```

شغّل الأمر `npm view [package_name] version` لمعرفة أحدث إصدار لحزمة في مستودع npm

```
npm view cowsay version
1.3.1
```

3.1.7 إلغاء تثبيت حزم npm

قد تسأل نفسك ماذا لو أردت إلغاء تثبيت حزمة npm المُثبَّتة تثبيتًا محليًا أو عامًا؟ يمكنك إلغاء تثبيت حزمة مثبَّتة مسبقًا محليًا locally باستخدام الأمر `npm install <packagename>` في مجلد `node_modules` بتشغيل الأمر التالي في مجلد جذر المشروع الذي يحتوي على مجلد `node_modules`:

```
npm uninstall <package-name>
```

تُستخدَم الراية `-S` أو `--save` لإزالة جميع المراجع في ملف package.json، فإذا كانت الحزمة عبارة عن اعتمادية تطوير مُدرَجة في اعتماديات devDependencies الخاصة بملف package.json، فيجب عليك استخدام الراية `-D` أو الراية `--save-dev` لإزالتها من الملف كما يلي:

```
npm uninstall -S <package-name>
npm uninstall -D <package-name>
```

إذا ثُبِّتت الحزمة تثبيتًا عامًا globally، فيجب إضافة الراية `-g` أو الراية `--global` كما يلي:

```
npm uninstall -g <package-name>
```

إليك المثال التالي:

```
npm uninstall -g webpack
```

كما يمكنك تشغيل هذا الأمر من أي مكان تريده على نظامك لأن المجلد الذي تتواجد فيه حاليًا غير مهم.

3.2 تشغيل مهام وتنفيذ سكربتات من سطر الأوامر

يدعم ملف package.json تنسيقًا لتحديد مهام سطر الأوامر التي يمكن تشغيلها باستخدام الأمر التالي:

```
npm run <task-name>
```

فمثلًا:

```
{
  "scripts": {
     "start-dev": "node lib/server-development",
     "start": "node lib/server-production"
  },
}
```

يشيع استخدام هذه الميزة لتشغيل Webpack:

```
{
  "scripts": {
     "watch": "webpack --watch --progress --colors --config
webpack.conf.js",
     "dev": "webpack --progress --colors --config webpack.conf.js",
     "prod": "NODE_ENV=production webpack -p --config
webpack.conf.js",
  },
}
```

يمكنك تشغيل الأوامر التالية بدلًا من كتابة الأوامر الطويلة السابقة التي يسهل نسيانها أو كتابتها بصورة خاطئة:

```
$ npm run watch
$ npm run dev
$ npm run prod
```

```
~/dev/node/cowsay/node_modules/.bin
❯ npx cowsay take me out of here
 _____________________
< take me out of here >
 ---------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

~/dev/node/cowsay/node_modules/.bin
❯
```

3.3 الملف package.json نقطة ارتكاز المشروع

يُعَدّ ملف package.json عنصرًا أساسيًا في كثير من قواعد شيفرات التطبيقات المستندة إلى نظام Node.js المجتمعي، فإذا استخدمت سابقًا لغة جافاسكريبت أو تعاملت مع مشروع JavaScript أو Node.js أو مشروع واجهة أمامية، فلا بد أنك صادفت ملف package.json، كما يُعَدّ ملف package.json بيانًا manifest لمشروعك، إذ يمكنه تطبيق أشياء غير مرتبطة متعددة، فهو مستودع مركزي لإعداد الأدوات مثلًا، كما أنه المكان الذي يخزّن فيه npm و yarn أسماء وإصدارات الحزم المُثبَّتة.

3.3.1 معمارية الملف package.json

فيما يلي مثال لملف package.json:

```
{
}
```

هذا الملف فارغ، إذ لا توجد متطلبات ثابتة لما يجب تواجده في ملف package.json خاص بتطبيقٍ ما، فالشرط الوحيد هو أنه يجب أن يتبع تنسيق JSON، وإلّا فلا يمكن أن تقرأه البرامج التي تحاول الوصول إلى خصائصه برمجيًا، وإذا أردت بناء حزمة Node.js التي ترغب في توزيعها عبر npm، فسيتغيّر كل شيء جذريًا، إذ

يجب أن يكون لديك مجموعة من الخصائص التي ستساعد الأشخاص الآخرين على استخدام هذا الملـف، حيث سنتحدّث عن ذلك لاحقًا، وإليك مثال آخر عن ملف package.json:

```
{
  "name": "test-project"
}
```

يعرِّف الملف السابق خاصية الاسم `name` والتي تعطي اسم التطبيق أو الحزمة الموجودة في المجلد نفسـه الذي يوجد فيه هذا الملف، وإليك المثال التالي الأكثر تعقيدًا والمُستخرَج من عينة تطبيق Vue.js:

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A Vue.js project",
  "main": "src/main.js",
  "private": true,
  "scripts": {
     "dev": "webpack-dev-server --inline --progress --config
build/webpack.dev.conf.js",
     "start": "npm run dev",
     "unit": "jest --config test/unit/jest.conf.js --coverage",
     "test": "npm run unit",
     "lint": "eslint --ext .js,.vue src test/unit",
     "build": "node build/build.js"
  },
  "dependencies": {
     "vue": "^2.5.2"
  },
  "devDependencies": {
     "autoprefixer": "^7.1.2",
     "babel-core": "^6.22.1",
     "babel-eslint": "^8.2.1",
     "babel-helper-vue-jsx-merge-props": "^2.0.3",
     "babel-jest": "^21.0.2",
     "babel-loader": "^7.1.1",
     "babel-plugin-dynamic-import-node": "^1.2.0",
     "babel-plugin-syntax-jsx": "^6.18.0",
```

```
        "babel-plugin-transform-es2015-modules-commonjs": "^6.26.0",
        "babel-plugin-transform-runtime": "^6.22.0",
        "babel-plugin-transform-vue-jsx": "^3.5.0",
        "babel-preset-env": "^1.3.2",
        "babel-preset-stage-2": "^6.22.0",
        "chalk": "^2.0.1",
        "copy-webpack-plugin": "^4.0.1",
        "css-loader": "^0.28.0",
        "eslint": "^4.15.0",
        "eslint-config-airbnb-base": "^11.3.0",
        "eslint-friendly-formatter": "^3.0.0",
        "eslint-import-resolver-webpack": "^0.8.3",
        "eslint-loader": "^1.7.1",
        "eslint-plugin-import": "^2.7.0",
        "eslint-plugin-vue": "^4.0.0",
        "extract-text-webpack-plugin": "^3.0.0",
        "file-loader": "^1.1.4",
        "friendly-errors-webpack-plugin": "^1.6.1",
        "html-webpack-plugin": "^2.30.1",
        "jest": "^22.0.4",
        "jest-serializer-vue": "^0.3.0",
        "node-notifier": "^5.1.2",
        "optimize-css-assets-webpack-plugin": "^3.2.0",
        "ora": "^1.2.0",
        "portfinder": "^1.0.13",
        "postcss-import": "^11.0.0",
        "postcss-loader": "^2.0.8",
        "postcss-url": "^7.2.1",
        "rimraf": "^2.6.0",
        "semver": "^5.3.0",
        "shelljs": "^0.7.6",
        "uglifyjs-webpack-plugin": "^1.1.1",
        "url-loader": "^0.5.8",
        "vue-jest": "^1.0.2",
        "vue-loader": "^13.3.0",
        "vue-style-loader": "^3.0.1",
```

```
        "vue-template-compiler": "^2.5.2",
        "webpack": "^3.6.0",
        "webpack-bundle-analyzer": "^2.9.0",
        "webpack-dev-server": "^2.9.1",
        "webpack-merge": "^4.1.0"
    },
    "engines": {
        "node": ">= 6.0.0",
        "npm": ">= 3.0.0"
    },
    "browserslist": [
        "> 1%",
        "last 2 versions",
        "not ie <= 8"
    ]
}
```

هناك خاصيات متعددة يجب شرحها في المثال السابق:

- name التي تضبط اسم التطبيق أو الحزمة.
- version التي تشير إلى الإصدار الحالي.
- description وهي وصف مختصَر للتطبيق أو للحزمة.
- main التي تضبط نقطة الدخول للتطبيق.
- private التي تمنع نشر التطبيق أو الحزمة عن طريق الخطأ على npm إذا ضُبِطت على القيمة true.
- scripts التي تحدّد مجموعة من سكربتات نود التي يمكنك تشغيلها.
- dependencies التي تضبط قائمة بحزم npm المثبَّتة كاعتماديات.
- devDependencies التي تضبط قائمة بحزم npm المثبَّتة كاعتماديات تطوير.
- engines التي تحدّد إصدار نود الذي تعمل عليه هذه الحزمة أو التطبيق.
- browserslist التي تُستخدَم لمعرفة المتصفحات وإصداراتها التي تريد دعمها.

تُستخدَم جميع هذه الخصائص إما باستخدام npm أو باستخدام أدوات أخرى.

3.3.2 خاصيات الملف package.json

يشرح هذا القسم الخاصيات التي يمكنك استخدامها ضمن الملف package.json بالتفصيل، حيث سنطبّق كل شيء على الحزمة، ولكن يمكن تطبيق الشيء نفسه على التطبيقات المحلية التي لا تستخدِمها على أساس حزم، كما تُستخدَم معظم هذه الخاصيات فقط على npm، ويُستخدَم البعض الآخر بواسطة السكربتات التي تتفاعل مع شيفرتك مثل npm أو غيره.

أ. name

تضبط هذه الخاصية اسم الحزمة مثل المثال التالي:

```
"name": "test-project"
```

يجب أن يتضمّن الاسم أقل من 214 محرفًا وألا يحتوي على مسافات، كما لا يمكن أن يحتوي إلّا على أحرف صغيرة أو واصلات أو شرطات سفلية _، وذلك لأن الحزمة تحصل على عنوان URL الخاص بها بناءً على هذه الخاصية عند نشرها على npm، إذا نشرتَ هذه الحزمة علنًا على GitHub، فستكون القيمة المناسبة لهذه الخاصية هي اسم مستودع GitHub.

ب. author

تعطي هذه الخاصية اسم مؤلف الحزمة كما في المثال التالي:

```
{
  "author": "Flavio Copes <flavio@flaviocopes.com>
(https://flaviocopes.com)"
}
```

يمكن استخدامها أيضًا بالتنسيق التالي:

```
{
  "author": {
     "name": "Flavio Copes",
     "email": "flavio@flaviocopes.com",
     "url": "https://flaviocopes.com"
  }
}
```

ج. contributors

يمكن أن يكون للمشروع مساهم أو أكثر بالإضافة إلى المؤلف، وهـذه الخاصـية هي مصـفوفة تعطي قائمـة المساهمين مثل المثال التالي:

```
{
  "contributors": [
    "Flavio Copes <flavio@flaviocopes.com> (https://flaviocopes.com)"
  ]
}
```

كما يمكن استخدام هذه الخاصية أيضًا بالتنسيق التالي:

```
{
  "contributors": [
    {
      "name": "Flavio Copes",
      "email": "flavio@flaviocopes.com",
      "url": "https://flaviocopes.com"
    }
  ]
}
```

د. bugs

تُستخدَم هذه الخاصية للربط بمتتبّع مشاكل الحزمة، أي بصفحة مشاكل GitHub مثلًا كما يلي:

```
{
  "bugs": "https://github.com/flaviocopes/package/issues"
}
```

ه. homepage

تضبط هذه الخاصية صفحة الحزمة الرئيسية كما في المثال التالي:

```
{
  "homepage": "https://flaviocopes.com/package"
}
```

و. version

تشير هذه الخاصية إلى إصدار الحزمة الحالي كما في المثال التالي:

```
"version": "1.0.0"
```

تتبع هذه الخاصية صيغة إدارة الإصدارات الدلالية semver، مما يعني أنّ الإصدار يُعبَّر عنه دائمًا بثلاثة أعداد: `x.x.x`، حيث يمثِّل العدد الأول الإصدار الرئيسي، ويمثِّل العدد الثاني الإصدار الثانوي؛ أما العدد الثالث فهو إصدار حزمة التصحيح patch version، فالإصدار الذي يصلح الأخطاء فقط هو إصدار حزمة التصحيح، والإصدار الذي يقدّم تغييرات متوافقة مع الإصدارات السابقة هو الإصدار الثانوي، كما يمكن أن يحتوي الإصدار الرئيسي على تغييرات جذرية.

ز. license

تشير إلى رخصة الحزمة مثل المثال التالي:

```
"license": "MIT"
```

ح. keywords

تحتوي هذه الخاصية على مصفوفة من الكلمات المفتاحية المرتبطة بما تفعله حزمتك مثل المثال التالي:

```
"keywords": [
  "email",
  "machine learning",
  "ai"
]
```

تساعد هذه الخاصية في العثور على حزمتك عند التنقل بين حزم مماثلة، أو عند تصفح موقع npm.

ط. description

تحتوي هذه الخاصية على وصف مختصَر للحزمة مثل المثال التالي:

```
"description": "A package to work with strings"
```

تفيد هذه الخاصية في حال قرّرت نشر حزمتك على `npm` وتوفير معلومات حول الحزمة.

ي. repository

تحدّد هذه الخاصية مكان وجود مستودع الحزمة كما في المثال التالي:

```
"repository": "github:flaviocopes/testing",
```

لاحظ البادئة `github`، وهناك خدمات شائعة أخرى مثل gitlab:

```
"repository": "gitlab:flaviocopes/testing",
```

وأيضًا bitbucket:

```
"repository": "bitbucket:flaviocopes/testing",
```

يمكنك ضبط نظام التحكم بالإصدارات بصورة صريحة كما يلي:

```
"repository": {
  "type": "git",
  "url": "https://github.com/flaviocopes/testing.git"
}
```

كما يمكنك استخدام أنظمة مختلفة للتحكم بالإصدارات كما يلي:

```
"repository": {
  "type": "svn",
  "url": "..."
}
```

ك. main

تضبط هذه الخاصية نقطة الدخول إلى الحزمة، وهي المكان الذي سيبحث فيه التطبيق عن عمليات تصدير الوحدة عند استيرادها في أحد التطبيقات مثل المثال التالي:

```
"main": "src/main.js"
```

ل. private

إذا ضُبِطت هذه الخاصية على القيمة `true`، فستمنع نشر التطبيق أو الحزمة عن طريق الخطأ على `npm` كما يلي:

```
"private": true
```

م. scripts

تحدّد هذه الخاصية مجموعة سكربتات نود التي يمكنك تشغيلها مثل المثال التالي:

```
"scripts": {
```

```
  "dev": "webpack-dev-server --inline --progress --config
build/webpack.dev.conf.js",
  "start": "npm run dev",
  "unit": "jest --config test/unit/jest.conf.js --coverage",
  "test": "npm run unit",
  "lint": "eslint --ext .js,.vue src test/unit",
  "build": "node build/build.js"
}
```

هذه السكربتات تطبيقات سطر الأوامر، ويمكنك تشغيلها عن طريـق اسـتدعاء الأمـر `npm run XXXX` أو `yarn XXXX`، حيث `XXXX` هو اسم الأمر مثل `npm run dev`، كما يمكنك إعطاء الأمر أيّ اسم تريده، ويمكن للسكربتات فعل أيّ شيء تريده.

ن. dependencies

تضبط هذه الخاصية قائمة حزم `npm` المثبَّتة على أساس اعتماديات. إذا ثبَّتَ حزمةً باستخدام `npm` أو `yarn` كما يلي:

```
npm install <PACKAGENAME>
yarn add <PACKAGENAME>
```

ستُدخَل تلك الحزمة في هذه القائمة تلقائيًا، وإليك المثال التالي:

```
"dependencies": {
  "vue": "^2.5.2"
}
```

س. devDependencies

تضبط هذه الخاصية قائمة حـزم `npm` المثبَّتـة على أسـاس اعتماديـات تطـوير، وهي تختلـف عن الخاصـية dependencies لأنهـا مُخصَّصـة للتثـبيت على آلـة تطـوير فقـط، وليسـت ضـرورية لتشـغيل الشـيفرة في عملية الإنتاج.

فإذا ثبَّتَ حزمةً باستخدام `npm` أو `yarn`:

```
npm install --dev <PACKAGENAME>
yarn add --dev <PACKAGENAME>
```

فستُدخَل تلك الحزمة في هذه القائمة تلقائيًا، وإليك المثال التالي:

```
"devDependencies": {
  "autoprefixer": "^7.1.2",
  "babel-core": "^6.22.1"
}
```

ع. engines

تضبط هذه الخاصية إصدارات Node.js والأوامر الأخرى التي تعمل عليها هذه الحزمة أو التطبيق كما في المثال التالي:

```
"engines": {
  "node": ">= 6.0.0",
  "npm": ">= 3.0.0",
  "yarn": "^0.13.0"
}
```

ف. browserslist

تُستخدَم هذه الخاصية لمعرفة المتصفحات وإصداراتها التي تريد دعمها، وقد أشارت إليها أدوات Babel و Autoprefixer وأدوات أخرى أنها تُستخدَم لإضافة تعويض نقص دعم المتصفحات polyfills والنسخ الاحتياطية fallbacks اللازمة للمتصفحات التي تستهدفها، وإليك المثال التالي:

```
"browserslist": [
  "> 1%",
  "last 2 versions",
  "not ie <= 8"
]
```

يعني الإعداد السابق أنك تريد دعم آخر إصدارين رئيسيين من جميع المتصفحات باستخدام 1% على الأقل -من إحصائيات موقع caniuse- باستثناء الإصدار IE8 والإصدارات الأقدم (اطلع على المزيد من موقع حزمة browserslist).

ص. خصائص خاصة بالأوامر

يمكن أن يستضيف ملف package.json أيضًا إعدادًا خاصًا بالأوامر مثل Babel و ESLint وغير ذلك، فلكل منها خاصية معينة مثل `eslintConfig` و `babel` وغيرها، وهذه هي الخصائص الخاصة بالأوامر، كما يمكنك العثور على كيفية استخدامها في توثيق الأمر أو المشروع المرتبط بها.

3.3.3 إصدارات الحزم

رأيت في الوصف أعلاه أرقام الإصدارات مثل: `3.0.0~` أو `0.13.0^`، حيث يحدِّد الرمز الموجود على يسار رقم الإصدار التحديثات التي تقبلها الحزمة من تلك الاعتمادية، ولنفترض استخدام semver -إدارة الإصدارات الدلالية semantic versioning-، حيث تتضمن جميع الإصدارات 3 خانات عددية، أولها هو الإصدار الرئيسي وثانيها هو الإصدار الثانوي وثالثها هو إصدار حزمة التصحيح patch release.

وبالتالي سيكون لديك القواعد التالية:

- `~`: إذا كتبت `0.13.0~`، فهذا يعني أنك تريد فقط تحديث إصدارات حزمة التصحيح، أي أنّ الإصدار `0.13.1` مقبول، ولكن الإصدار `0.14.0` ليس كذلك.
- `^`: إذا كتبت `0.13.0^`، فهذا يعني أنك تريد تحديث إصدار حزمة التصحيح والإصدار الثانوي، أي الإصدارات `0.13.1` و `0.14.0` وهكذا.
- `*`: إذا كتبت `*`، فهذا يعني أنك تقبل جميع التحديثات بما في ذلك ترقيات الإصدارات الرئيسية.
- `>`: أي أنك تقبل أي إصدار أعلى من الإصدار الذي تحدِّده.
- `>=`: أي أنك تقبل أي إصدار مساوي أو أعلى من الإصدار الذي تحدّده.
- `<=`: أي أنك تقبل أي إصدار مساوي أو أدنى من الإصدار الذي تحدِّده.
- `<`: أي أنك تقبل أي إصدار أدنى من الإصدار الذي تحدِّده.

وهناك قواعد أخرى هي:

- بدون رمز: أي أنك تقبل فقط الإصدار الذي تحدّده.
- `latest`: أي أنك تريد استخدام أحدث إصدار متاح.

كما يمكنك دمج معظم ما سبق ضمن مجالات مثل `<1.2.0 >=1.1.0 || 1.0.0` لاستخدام إما الإصدار `1.0.0` أو أحد الإصدارات الأعلى أو المساوية للإصدار `1.1.0` والأدنى من الإصدار `1.2.0`.

3.4 الملف package-lock.json ودوره في إدارة الإصدارات

يُنشَأ الملف package-lock.json تلقائيًا عند تثبيت حزم نود، حيث قدَّم npm ملف `package-lock.json` في الإصدار رقم 5، والهدف من هذا الملف هو تتبّع الإصدار الدقيق لكل حزمة مثبَّتة، وبالتالي فإن المنتج قابل لإعادة الإنتاج بنسبة 100% بالطريقة نفسها حتى إذا حدّث القائمون على الصيانة الحزم، كما يحل ذلك مشكلةً تركَها ملف `package.json` دون حل.

إذ يمكنك في ملف `package.json` ضبط الإصدارات التي تريد الترقية إليها -أي إصدار حزمة التصحيح أو الإصدار الثانوي- باستخدام صيغة semver كما يلي:

- إذا كتبت `~0.13.0`، فهذا يعني أنك تريد فقط تحديث إصدار حزمة التصحيح، أي أن الإصدار `0.13.1` مقبول، ولكن الإصدار `0.14.0` ليس كذلك.
- إذا كتبت `^0.13.0`، فهذا يعني أنك تريد تحديث إصدار حزمة التصحيح والإصدار الثانوي، أي الإصدارات `0.13.1` و `0.14.0` وهكذا.
- إذا كتبت `0.13.0`، فهذا يعني الإصدار الدقيق الذي سيُستخدَم.

لستَ ملزَمًا بتوزيع مجلد `node_modules` الضخم باستخدام Git، وإذا حاولتَ نسخ المشروع على جهاز آخر باستخدام الأمر `npm install` -إذا حدّدتَ الصيغة ~ مع إصدار حزمة التصحيح الخاص بالحزمة- فسيثبَّت هذا الإصدار، كما يحدث الأمر ذاته مع الصيغة ^ والإصدارات الثانوية.

> إذا حدّدتَ إصدارات معينة مثل الإصدار `0.13.0`، فلن تتأثر بهذه المشكلة.

قد تحاول أنت أو أي شخص آخر تهيئة المشروع على الجانب الآخر من العالم عن طريق تشغيل الأمر `npm install`، لذلك فإن مشروعك الأصلي والمشروع المُهيَّأ حديثًا مختلفان فعليًا، إذ يجب ألّا يدخِل إصدار حزمة التصحيح أو الإصدار الثانوي تغييرات معطِّلة، ولكننا نعلم أن الأخطاء ممكنة الحدوث وستحدث بالفعل.

يضبط ملف `package-lock.json` الإصدار المثبَّت حاليًا من كل حزمة باستخدام رمز stone، وسيستخدِم `npm` هذه الإصدارات المحدَّدة عند تشغيل الأمر `npm install`، كما أنّ هذا المفهوم ليس بجديد، إذ يستخدِم مديرو حزم لغات البرمجة الأخرى -مثل مكتبات Composer في لغة PHP- نظامًا مشابهًا منذ سنوات.

يجب أن يكون ملف `package-lock.json` ملتزمًا بمستودع Git الخاص بك حتى يجلبه أشخاص آخرون إذا كان المشروع عامًا أو لديك متعاونون أو إذا استخدمت Git على أساس مصدر لعمليات النشر، كما ستُحدَّث إصدارات الاعتماديات في ملف `package-lock.json` عند تشغيل الأمر `npm update`.

يوضِّح المثال التالي معمارية ملف `package-lock.json` التي سنحصل عليها عند تشغيل الأمر `npm install cowsay` في مجلد فارغ:

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
     "ansi-regex": {
```

```
        "version": "3.0.0",
        "resolved": "https://registry.npmjs.org/ansi-regex/-/ansi-
regex-3.0.0.tgz",
        "integrity": "sha1-7QMXwyIGT3lGbAKWa922Bas32Zg="
  },
  "cowsay": {
     "version": "1.3.1",
     "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-
1.3.1.tgz",
     "integrity":
"sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BTTDkMAjufp+OF9eLjzRnOH
zVAYeIYFF5po5NjRrgefnRMQ==",
     "requires": {
        "get-stdin": "^5.0.1",
        "optimist": "~0.6.1",
        "string-width": "~2.1.1",
        "strip-eof": "^1.0.0"
     }
  },
  "get-stdin": {
     "version": "5.0.1",
     "resolved": "https://registry.npmjs.org/get-stdin/-/get-stdin-
5.0.1.tgz",
     "integrity": "sha1-Ei4WFZHiH/TFJTAwVpPyDmOTo5g="
  },
  "is-fullwidth-code-point": {
     "version": "2.0.0",
     "resolved": "https://registry.npmjs.org/is-fullwidth-code-
point/-/is-fullwidth-code-point-2.0.0.tgz",
     "integrity": "sha1-o7MKXE8ZkYMWeqq5O+764937ZU8="
  },
  "minimist": {
     "version": "0.0.10",
     "resolved": "https://registry.npmjs.org/minimist/-/minimist-
0.0.10.tgz",
     "integrity": "sha1-3j+YVD2/lggr5IrRoMfNqDYwHc8="
  },
  "optimist": {
```

```
      "version": "0.6.1",
      "resolved": "https://registry.npmjs.org/optimist/-/optimist-
0.6.1.tgz",
      "integrity": "sha1-2j6nRob6IaGaERwybpDrFaAZZoY=",
      "requires": {
        "minimist": "~0.0.1",
        "wordwrap": "~0.0.2"
      }
    },
    "string-width": {
      "version": "2.1.1",
      "resolved": "https://registry.npmjs.org/string-width/-/string-
width-2.1.1.tgz",
      "integrity": "sha512-
nOqH59deCq9SRHlxq1Aw85Jnt4w6KvLKqWVik6oA9ZklXLNIOlqg4F2yrT1MVaTjAqvVwd
feZ7w7aCvJD7ugkw==",
      "requires": {
        "is-fullwidth-code-point": "^2.0.0",
        "strip-ansi": "^4.0.0"
      }
    },
    "strip-ansi": {
      "version": "4.0.0",
      "resolved": "https://registry.npmjs.org/strip-ansi/-/strip-ansi-
4.0.0.tgz",
      "integrity": "sha1-qEeQIusaw2iocTibY1JixQXuNo8=",
      "requires": {
        "ansi-regex": "^3.0.0"
      }
    },
    "strip-eof": {
      "version": "1.0.0",
      "resolved": "https://registry.npmjs.org/strip-eof/-/strip-eof-
1.0.0.tgz",
      "integrity": "sha1-u0P/VZim6wXYm1n80SnJgzE2Br8="
    },
    "wordwrap": {
      "version": "0.0.3",
```

```
      "resolved": "https://registry.npmjs.org/wordwrap/-/wordwrap-
0.0.3.tgz",
      "integrity": "sha1-o9XabNXAvAAI03IOu68b7WMFkQc="
      }
    }
  }
```

ثبّتنا حزمة cowsay التي تعتمد على الحزم التالية:

- get-stdin.
- optimist.
- string-width.
- strip-eof.

تتطلب هذه الحزم حزمًا أخرى مثل الحزم الموجودة في الخاصية requires كما يلي:

- ansi-regex.
- is-fullwidth-code-point.
- minimist.
- wordwrap.
- strip-eof.

تُضاف هذه الحزم إلى الملف بالترتيب الأبجدي، ولكل منها حقل version، وحقل resolved يؤشّر إلى موقع الحزمة، وسلسلة نصية integrity يمكننا استخدامها للتحقق من الحزمة.

3.5 قواعد الإدارة الدلالية لنسخ الاعتماديات

تُعَدّ الإدارة الدلالية للإصدارات Semantic Versioning اصطلاحًا يُستخدَم لتوفير معنى للإصدارات، فإذا كان هناك شيء رائع في حزم Node.js، فهو اتفاق الجميع على استخدام هذا المفهوم لترقيم إصداراتهم، كما يُعَدّ مفهوم الإدارة الدلالية للنسخ بسيطًا للغاية، فلكل الإصدارات 3 خانات عددية x.y.z:

- العدد الأول هو الإصدار الرئيسي.
- العدد الثاني هو الإصدار الثانوي.
- العدد الثالث هو إصدار التصحيح.

إذا أردت إنشاء إصدار جديد، فلن تزيد عددًا كما يحلو لك، بل لديك قواعد يجب الالتزام بها وهي:

- يُحدَّث الإصدار الرئيسي عند إجراء تغييرات غير متوافقة مع واجهة برمجة التطبيقات API.
- يُحدَّث الإصدار الثانوي عند إضافة عمليات بطريقة متوافقة مع الإصدارات السابقة.
- يُحدَّث إصدار تصحيح عند إجراء إصلاحات أخطاء متوافقة مع الإصدارات السابقة.

أُعتمِد هذا المفهوم في جميع لغات البرمجة ومن المهم أن تلتزم بها كل حزمة npm لأن النظام بأكمله يعتمد على ذلك، إذ وضَع npm بعض القواعد التي يمكننا استخدامها في ملف `package.json` لاختيار الإصدارات التي يمكن تحديث حزمنا إليها عند تشغيل الأمر `npm update`، وتستخدِم هذه القواعد الرموز التالية:

- `^`: إذا كتبت `^0.13.0` عند تشغيل الأمر `npm update`، فهذا يؤدي إلى تحديث إصدار حزمة التصحيح والإصدار الثانوي، أي الإصدارات `0.13.1` و `0.14.0` وهكذا.
- `~`: إذا كتبت `~0.13.0` عند تشغيل الأمر `npm update`، فهذا يؤدي إلى تحديث إصدارات حزمة التصحيح، أي أن الإصدار `0.13.1` مقبول، ولكن الإصدار `0.14.0` ليس كذلك.
- `>`: أي أنك تقبل أي إصدار أعلى من الإصدار الذي تحدده.
- `>=`: أي أنك تقبل أي إصدار مساوي أو أعلى من الإصدار الذي تحدّده.
- `<=`: أي أنك تقبل أي إصدار مساوي أو أدنى من الإصدار الذي تحدّده.
- `<`: أي أنك تقبل أي إصدار أدنى من الإصدار الذي تحدّده.
- `=`: أي أنك تقبل الإصدار المحدَّد.
- `-`: أي أنك تقبل مجالًا من الإصدارات مثل المجال مثل: `2.1.0 - 2.6.2`.
- `||`: يُستخدَم لدمج مجموعات من الإصدارات مثل: `< 2.1 || > 2.6`.

يمكنك دمج بعض القواعد السابقة مثل: `1.0.0 || >=1.1.0 <1.2.0` لاستخدام إما الإصدار `1.0.0` أو أحد الإصدارات الأعلى أو المساوية للإصدار `1.1.0` والأدنى من الإصدار `1.2.0`.

هناك قواعد أخرى أيضًا هي:

- بدون رمز: أي أنك تقبل فقط الإصدار الذي تحدّده مثل: `1.2.1`.
- `latest`: أي أنك تريد استخدام أحدث إصدار متاح.

3.6 أنواع الحزم

تُصنَّف الحزم وفقًا لمجال نطاق رؤيتها، أي المكان الذي تُرَى الحزمة منه ويمكن استخدامها فيه، وتنقسم إلى حزمة عامة وخاصة أو محلية، كما تصنَّف أيضًا وفقًا لبيئة استخدامها وتكون إما اعتماديات أساسية ضرورية للمشروع في بيئة الإنتاج والتطوير معًا، وإما اعتماديات خاصة ببيئة التطوير فقط، أي مطلوبة في وقت تطوير المشروع وغير مطلوبة في بيئة الإنتاج.

3.6.1 الحزم العامة والحزم المحلية

الفرق الرئيسي بين الحزم المحلية والعامة هو:

- تُثبَّت الحزم المحلية في المجلد أو المسار حيث تشغّل الأمر `npm install <package-name>`، وتوضَع في مجلد `node_modules` ضمن هذا المجلد أو المسار.
- توضَع جميع الحزم العامة في مكان واحد في نظامك بالاعتماد على إعدادك الخاص بغض النظر عن مكان تشغيل الأمر `npm install -g <package-name>`.

وكلاهما مطلوب بالطريقة نفسها في شيفرتك الخاصة كما يلي:

```
require('package-name')
```

يجب تثبيت جميع الحزم محليًا، إذ يضمن ذلك أنه يمكنك الحصول على عشرات التطبيقات على حاسوبك، وتشغِّل جميعها إصدارًا مختلفًا من كل حزمة إذ لزم الأمر، بينما سيجعل تحديث حزمة عامة جميع مشاريعك تستخدِم الإصدار الجديد، مما قد يسبّب مشكلات ضخمة في عملية الصيانة، حيث قد تخرِّب بعض الحزم التوافق بمزيد من الاعتماديات وما إلى ذلك.

تحتوي جميع المشاريع على نسختها المحلية الخاصة من الحزمة، فقد يبدو ذلك ضياعًا للموارد، ولكنه ضياع ضئيل بالموازنة مع العواقب السلبية المحتمَلة، كما يجب تثبيت الحزمة العامة عندما توفِّر هذه الحزمة أمرًا قابلًا للتنفيذ بحيث تشغّله من الصدفة shell أي واجهة سطر الأوامر CLI، ويُعاد استخدام هذه الحزمة عبر المشاريع، كما يمكنك أيضًا تثبيت الأوامر القابلة للتنفيذ محليًا وتشغيلها باستخدام `npx`، ولكن تثبيت الحزم العامة أفضل بالنسبة لبعض الحزم.

فيما يلي أمثلة رائعة عن الحزم العامة الشائعة التي قد تعرفها:

- `npm`
- `create-react-app`
- `vue-cli`
- `grunt-cli`

- `mocha`
- `react-native-cli`
- `gatsby-cli`
- `forever`
- `nodemon`

يُحتمَل أن تكون لديك بعض الحزم العامة المثبَّتة على نظامك التي يمكنك رؤيتها عن طريق تشغيل الأمر التالي في سطر الأوامر الخاص بك:

```
npm list -g --depth 0
```

3.6.2 الاعتماديات الأساسية واعتماديات التطوير

إذا ثبّتَ حزمة `npm` باستخدام الأمر `npm install <package-name>`، فهذا يعني أنك ثبّتها على أساس اعتمادية dependency، حيث تُدرَج الحزمة تلقائيًا في ملف `package.json` ضمن قائمة `dependencies` بدءًا من الإصدار `npm 5`، إذ احتجنا سابقًا إلى تحديد الراية `--save` يدويًا، فإذا أضفتَ الراية `-D` أو الراية `--save-dev`، فهذا يعني أنك تثبّتها على أساس اعتمادية تطوير، وبالتالي ستُضاف إلى قائمة `devDependencies`.

يُقصَد باعتماديات التطوير أنها حزم للتطوير فقط، وهي غير ضرورية في عملية الإنتاج مثل حزم الاختبار أو حزم `webpack` أو `Babel`، فإذا كتبت الأمر `npm install` واحتوى المجلد على ملف `package.json` في عملية الإنتاج، فستُثبَّت اعتماديات التطوير، حيث يفترض `npm` أنّ هذه عملية نشر تطوير، كما يجب ضبط الراية `--production` من خلال الأمر `npm install --production` لتجنب تثبيت اعتماديات التطوير.

3.7 أداة تشغيل الشيفرة npx

يُعَدّ npx طريقةً رائعةً جدًا لتشغيل شيفرة نود، كما يوفِّر ميزات مفيدةً متعددةً، إذ كان متاحًا في `npm` بدءًا من الإصدار 5.2، الذي صدر في شهر 7 من عام 2017.

> إذا لم ترغب في تثبيت npm، فيمكنك تثبيت npx على أساس حزمة قائمة بذاتها.

يتيح لك `npx` تشغيل الشيفرة المُنشَأة باستخدام نود والمنشورة من خلال سجل `npm`، ويتميز `npx` بالمميزات التالية:

3.7.1 تشغيل الأوامر المحلية بسهولة

اعتاد مطورو نود على نشر معظم الأوامر القابلة للتنفيذ على أساس حزم عامة لتكون هذه الأوامر في المسار الصحيح وقابلةً للتنفيذ مباشرةً، إذ كان هذا أمرًا صعبًا جدًا، لأن تثبيت إصدارات مختلفة من الأمر نفسه غير ممكن، كما يؤدي تشغيل الأمر `npx commandname` تلقائيًا إلى العثور على مرجع الأمر الصحيح ضمن مجلد `node_modules` الخاص بالمشروع دون الحاجة إلى معرفة المسار الدقيق، ودون الحاجة إلى تثبيت الحزمة على أنها حزمة عامة وفي مسار المستخدِم.

3.7.2 تنفيذ الأوامر دون تثبيتها

هناك ميزة أخرى رائعة في `npm` تسمح بتشغيل الأوامر دون تثبيتها أولًا، حيث يُعَدّ ذلك مفيدًا جدًا للأسباب التالية:

1. لا تحتاج إلى تثبيت أي شيء.
2. يمكنك تشغيل إصدارات مختلفة من الأمر نفسه باستخدام صيغة `@version`.

يمكن توضيح استخدام `npx` من خلال الأمر `cowsay` الذي سيطبع بقرةً تقول ما تكتبه ضمن الأمر، حيث سيطبع الأمر `cowsay "Hello"` ما يلي على سبيل المثال:

```
 _______
< Hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

يحدث ذلك إذا كان الأمر `cowsay` مثبَّتًا تثبيتًا عامًا من `npm` سابقًا، وإلا فستحصل على خطأ عند محاولة تشغيل الأمر، كما يسمح لك `npx` بتشغيل الأمر السابق دون تثبيته محليًا كما يلي:

```
npx cowsay "Hello"
```

يُعَدّ الأمر السابق للتسلية فقط ودون فائدة، ولكن يمكنك استخدام `npx` في حالات مهمة أخرى مثل:

- تشغيل أداة واجهة سطر الأوامر `vue` لإنشاء تطبيقات جديدة وتشغيلها باستخدام الأمر `npx vue create myvue-app`
- إنشاء تطبيق React جديد باستخدام الأمر `npx create-react-app my-react-app`.

و حالات أخرى أيضًا، كما ستُمسَح الشيفرة المُنزَّلة لهذه الأوامر بمجرد تنزيلها.

3.7.3 تشغيل شيفرة باستخدام إصدار نود Node مختلف

استخدم الرمز @ لتحديد الإصدار، وادمج ذلك مع حزمة npm التي هي node:

```
npx node@6 -v #v6.14.3
npx node@8 -v #v8.11.3
```

يساعد ذلك في تجنب استخدام أدوات مثل أداة nvm أو أدوات إدارة إصدارات نود الأخرى.

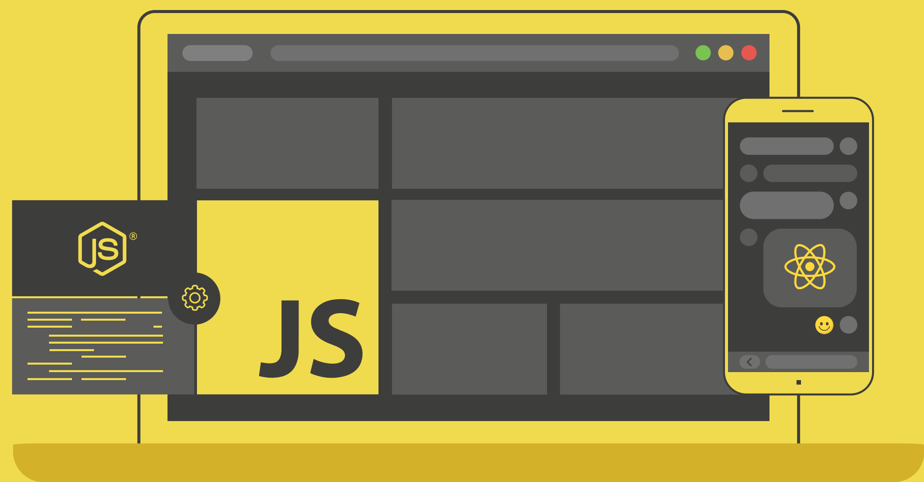
3.7.4 تشغيل أجزاء شيفرة عشوائية مباشرة من عنوان URL

لا يقيّدك npx بالحزم المنشورة في سجل npm، إذ يمكنك تشغيل الشيفرة الموجودة في GitHub gist كما في المثال التالي:

```
npx https://gist.github.com/zkat/4bc19503fe9e9309e2bfaa2c58074d32
```

يجب أن تكون حذرًا عند تشغيل شيفرة لا تتحكم بها، فالقوة العظمى تستوجب مسؤولية عظمى أيضًا.

دورة تطوير التطبيقات باستخدام لغة JavaScript



احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقًا من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن

4. كيفية تنفيذ الدوال داخليا ضمن Node.js

ستتعرّف في هذا الفصل على مفهوم حلقة الأحداث وكيفية سير عملية تنفيذ الدوال تنفيذًا غير متزامن ضمن Node.js، كما سنوضّح كيفية التعامل مع الأحداث المخصّصة من خلال الصنف `EventEmitter` المضمن في نود والذي يُستخدَم لمعالجة الأحداث.

4.1 حلقة الأحداث event loop

تُعَدّ حلقة الأحداث Event Loop أحد أهم جوانب جافا سكريبت التي يجب فهمها.

> قد تكون مبرمج جافا سكريبت منذ سنوات، ولكنك لم تفهم تمامًا كيفية سير الأمور الداخلية، إذ لا يُعَدّ عدم معرفتك بهذا المفهوم بالتفصيل عيبًا، ولكن معرفتك بذلك أمر جيد.

4.1.1 مدخل إلى حلقة الأحداث

سنشرح التفاصيل الداخلية لكيفية عمل جافا سكريبت باستخدام خيط thread واحد، وسنوضّح كيفية معالجة الدوال غير المتزامنة.

تُشغَّل شيفرة جافا سكريبت الخاصة بك ضمن خيط واحد، أي أن هناك شيئًا واحدًا فقط يحدث في الوقت نفسه، هذا القيد مفيد جدًا لأنه يبسّط كثيرًا من عملية البرمجة دون القلق بشأن مشاكل التزامن، فما عليك إلا التركيز على كيفية كتابة شيفرتك الخاصة وتجنب أي شيء يمكن إيقاف الخيط مثل استدعاءات الشبكة المتزامنة أو الحلقات اللانهائية.

توجد حلقة أحداث لكل تبويب في معظم المتصفحات لعزل العمليات عن بعضها البعض وتجنب صفحة الويب ذات الحلقات اللانهائية أو ذات المعالجة الكبيرة التي تؤدي إلى توقّف المتصفح بأكمله، كما تدير البيئة حلقات أحداث متزامنة متعددة لمعالجة استدعاءات واجهة API مثلًا، كما تُشغَّل عمّال الويب Web Workers

في حلقة الأحداث الخاصة بها أيضًا، إذ يجب عليك الاهتمام فقط بتشغيل شيفرتك ضمن حلقة أحداث واحدة، وكتابة شيفرتك مع وضع ذلك في الحسبان لتجنب توقفها.

4.1.2 إيقاف حلقة الأحداث

ستوقِف شيفرة جافا سكريبت التي تستغرق وقتًا طويلًا لإعادة التحكم إلى حلقة الأحداث مرةً أخرى تنفيذَ أيّ شيفرة جافا سكريبت في الصفحة، إذ يمكن أن توقِف خيط واجهة المستخدِم، وبالتالي لا يمكن للمستخدِم تمرير الصفحة أو النقر عليها وغير ذلك، كما تُعَدّ جميع عناصر الدخل أو الخرج الأولية في جافا سكريبت غير قابلة للإيقاف non-blocking تقريبًا مثل طلبات الشبكة وعمليات نظام ملفات Node.js وما إلى ذلك، ولكن الاستثناء هو توقّفها، وهذا هو سبب اعتماد جافا سكريبت الكبير على دوال رد النداء callbacks واعتمادها مؤخرًا على الوعود promises وصيغة عدم التزامن أو الانتظار async/await.

4.1.3 مكدس الاستدعاءات call stack

مكدس الاستدعاءات هو طابور LIFO أي القادم أخيرًا يخرج أولًا Last In First Out، حيث تتحقّق حلقة الأحداث باستمرار من مكدس الاستدعاءات للتأكد من وجود دالة يجب تشغيلها، حيث تضيف حلقة الأحداث عندها أي استدعاء دالة تجده إلى مكدس الاستدعاءات وتنفّذ كل استدعاء بالترتيب.

قد تكون على دراية بتعقّب مكدس الأخطاء في منقّح الأخطاء debugger أو في وحدة تحكم المتصفح، حيث يبحث المتصفح عن أسماء الدوال في مكدس الاستدعاءات لإعلامك بالدالة التي تنشئ الاستدعاء الحالي:

```
> const bar = () => {
    throw new DOMException()
  }

  const baz = () => console.log('baz')

  const foo = () => {
    console.log('foo')
    bar()
    baz()
  }

  foo()
  foo
✖ ▼Uncaught DOMException
    bar          @ VM570:2
    foo          @ VM570:9
    (anonymous) @ VM570:13
> |
```

4.1.4 شرح بسيط لحلقة الأحداث

افترض المثال التالي:

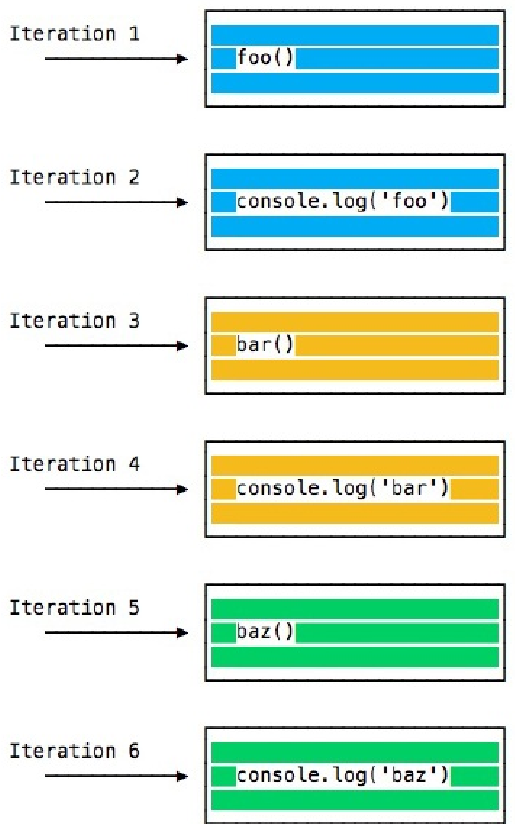
```
const bar = () => console.log('bar')
const baz = () => console.log('baz')
const foo = () => {
 console.log('foo')
 bar()
 baz()
}
foo()
```

الذي يطبع ما يلي:

```
foo
bar
baz
```

تُستدعَى الدالة `foo()` أولًا عند تشغيل الشيفرة السابقة، ثم نستدعي الدالة `bar()` أولًا ضمن الدالة `foo()`، ثم نستدعي الدالة `baz()`، ويبدو مكدس الاستدعاءات في هذه المرحلة كما يلي:

تتأكد حلقة الأحداث في كل تكرار من وجود شيء ما في مكدس الاستدعاءات، وتنفِّذه كما يلي إلى أن يصبح مكدس الاستدعاءات فارغًا:



4.1.5 تنفيذ طابور الدوال

لا يوجد شيء مميز في المثال السابق، حيث تعثر شيفرة جافا سكريبت على الدوال لتنفيذها وتشغيلها بالترتيب، ولنشاهد كيفية تأجيل تنفيذ دالة إلى أن يصبح المكدس فارغًا، حيث تُستخدَم حالة الاستخدام `setTimeout(() => {}), 0)` لاستدعاء دالة، ولكنها تُنفَّذ عند كل تنفيذ لدالة أخرى في الشيفرة.

إليك المثال التالي:

```
const bar = () => console.log('bar')
const baz = () => console.log('baz')
const foo = () => {
 console.log('foo')
 setTimeout(bar, 0)
 baz()
}
foo()
```

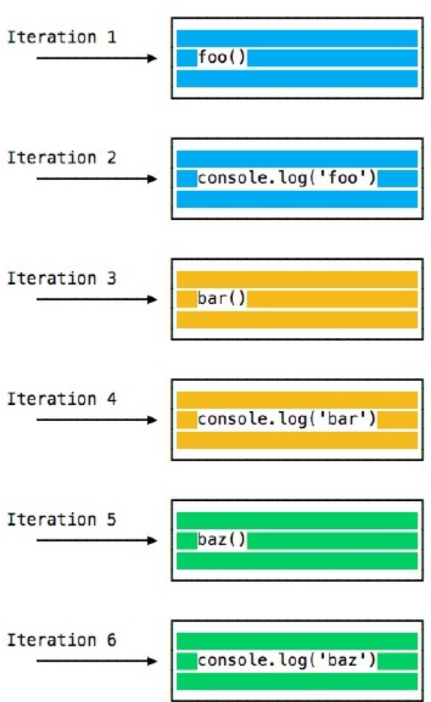
تطبع الشيفرة السابقة ما يلي:

```
foo
baz
bar
```

تُستدعَى الدالة `foo()` أولًا عند تشغيل الشيفرة، ثم نستدعي `setTimeout` أولًا ضمن الدالة `foo()`، ونمرّر `bar` على أساس وسيط، ونطلب منه العمل على الفور بأسرع ما يمكنه، ونمرر القيمة `0` على أساس مؤقت timer، ثم نستدعي الدالة `baz()`.

يبدو مكدس الاستدعاءات في هذه المرحلة كما يلي:

يوضِّح الشكل التالي ترتيب تنفيذ جميع الدوال في البرنامج:



4.1.6 طابور الرسائل Message Queue

يبدأ المتصفح أو Node.js المؤقت timer عند استدعاء الدالة `setTimeout()`، ثم توضَع دالة رد النداء callback function في طابور الرسائل Message Queue بمجرد انتهاء صلاحية المؤقت على الفور مثل حالة وضع القيمة 0 على أساس مهلة زمنية timeout.

يُعَدّ طابور الرسائل المكان الذي توضَع الأحداث التي بدأها المستخدِم مثل أحداث النقر، أو أحداث لوحة المفاتيح، أو جلب الاستجابات الموجودة في طابور قبل أن تتاح لشيفرتك فرصة الرد عليها، أو أحداث DOM مثل `onLoad`.

> تعطِي الحلقة الأولوية لمكدّس الاستدعاءات، وتعالج كل شيء تجده فيه أولًا، ثم تنتقل لالتقاط الأشياء الموجودة في طابور الأحداث عند عدم وجود أي شيء في مكدّس الاستدعاءات.

لا يتعيّن علينا انتظار دوال مثل الدالة `setTimeout` أو انتظار جلب أو تنفيذ أشياء أخرى لهذه الدوال لأن المتصفح يوفّرها وتتقيّد بخيوطها الخاصة، فإذا ضبطتَ مهلة `setTimeout` الزمنية على 2 ثانية مثلًا، فلن تضطر إلى الانتظار لمدة 2 ثانية، بل يحدث الانتظار في مكان آخر.

4.1.7 طابور العمل Job Queue الخاص بالإصدار ES6

قدّم المعيار ECMAScript 2015 مفهوم طابور العمل Job Queue الذي تستخدمه الوعود Promises التي قُدِّمت أيضًا ضمن الإصدار ES6/ES2015، ويُعَدّ هذا المفهوم طريقةً لتنفيذ نتيجة دالة غير متزامنة بأسرع ما يمكن بدلًا من وضعها في نهاية مكدس الاستدعاءات.

ستُنفَّذ الوعود المؤكَّدة قبل انتهاء الدالة الحالية بعدها مباشرةً، حيث يشبه ذلك ركوب الأفعوانية في مدينة ملاهي، إذ يضعك طابور الرسائل بعد جميع الأشخاص الآخرين الموجودين في هذا الطابور، بينما طابور العمل هو مثل تذكرة Fastpass تتيح لك الركوب في رحلة أخرى في الأفعوانية بعد الانتهاء من الرحلة السابقة مباشرةً.

إليك المثال التالي:

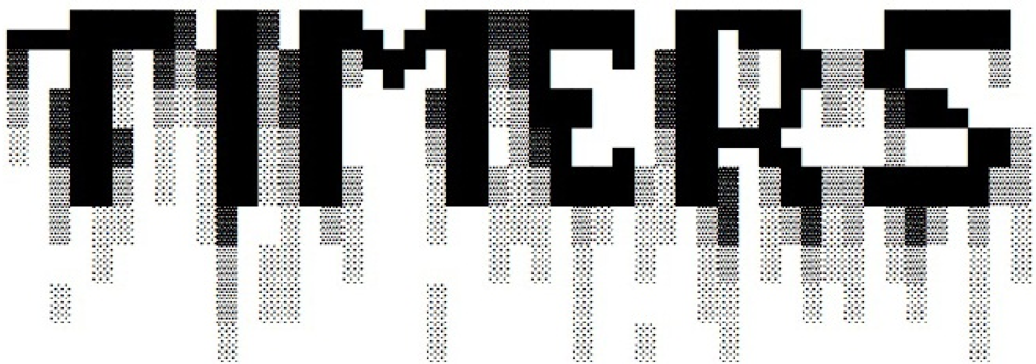
```
const bar = () => console.log('bar')
const baz = () => console.log('baz')
const foo = () => {
 console.log('foo')
 setTimeout(bar, 0)
 new Promise((resolve, reject) =>
   resolve('should be right after baz, before bar')).then(resolve =>
console.log(resolve))
 baz()
}
foo()
```

تطبع الشيفرة السابقة ما يلي:

```
foo
baz
should be right after baz, before bar
bar
```

يشكّل ذلك فرقًا كبيرًا بين الوعود Promises وصيغة Async/await المبنيّة على الوعود والدوال القديمة غير المتزامنة من خلال الدالة `setTimeout()` أو واجهات API للمنصات الأخرى.

4.2 المؤقتات Timers: التنفيذ غير المتزامن في أقرب وقت ممكن



تُعَدّ الدالة `process.nextTick()` جزءًا مهمًا من حلقة أحداث Node.js، حيث نسمّي كل دورة كاملة تدورها حلقة الأحداث بالاسم نبضة tick، كما يؤدي تمرير دالة إلى `process.nextTick()` إلى استدعاء هذه الدالة في نهاية العملية الحالية وقبل بدء نبضة حلقة الأحداث التالية.

```
process.nextTick(() => {
  //افعل شيئًا ما
})
```

حلقة الأحداث مشغولة بمعالجة شيفرة الدالة الحالية، ويشغّل محرك JS عند انتهاء هذه العملية جميع الدوال المُمرَّرة إلى استدعاءات `nextTick` خلال تلك العملية، وهذه هي الطريقة التي يمكننا من خلالها إخبار محرك JS بمعالجة دالة بطريقة غير متزامنة بعد الدالة الحالية في أقرب وقت ممكن دون وضعها في طابور، كما سيؤدي استدعاء `setTimeout(() => {}, 0)` إلى تنفيذ الدالة في النبضة التالية بعد وقت أطول من استخدام الدالة `nextTick()`، واستخدم الدالة `nextTick()` عندما تريد التأكد من تنفيذ الشيفرة في تكرار حلقة الأحداث التالي.

4.2.1 الدالة setTimeout()

قد ترغب في تأخير تنفيذ دالة عند كتابة شيفرة جافا سكريبت، وهذه هي مهمة الدالة `setTimeout`، حيث تحدِّد دالة رد نداء لتنفيذها لاحقًا مع قيمة تعبّر عن مقدار التأخير لتشغيلها لاحقًا مقدَّرةً بالميلي ثانية:

```
setTimeout(() => {
  // تشغيل بعد 2 ثانية
}, 2000)
setTimeout(() => {
  // تشغيل بعد 50 ميلي ثانية
}, 50)
```

تحدِّد هذه الصيغة دالةً جديدةً، حيث يمكنك استدعاء أيّ دالة أخرى تريدها هناك، أو يمكنك تمرير اسم دالة موجودة مسبقًا مع مجموعة من المعاملات كما يلي:

```
const myFunction = (firstParam, secondParam) => {
  // افعل شيئًا ما
}
// تشغيل بعد 2 ثانية
setTimeout(myFunction, 2000, firstParam, secondParam)
```

تعيد الدالة `setTimeout` معرِّف المؤقت `id`، وهذا المعرِّف غير مُستخدَم، ولكن يمكنك تخزينه ومسحه إذا أردت حذف تنفيذ الدوال المجدولة:

```
const id = setTimeout(() => {
  // يجب تشغيله بعد 2 ثانية
}, 2000)
//غيّرنا رأينا
clearTimeout(id)
```

4.2.2 الدالة setImmediate

إذا أردت تنفيذ جزء من الشيفرة بطريقة غير متزامنة ولكن في أقرب وقت ممكن، فإنّ أحد الخيارات هو استخدام الدالة `setImmediate()` التي يوفّرها Node.js:

```
setImmediate(() => {
  //شغّل شيئًا ما
})
```

تمثِّل الدالة المُرَّرة على أساس وسيط للدالة `setImmediate()` دالّةَ رد نداء تُنفَّذ في التكرار التالي لحلقة الأحداث، كما تختلف `setImmediate()` عن `setTimeout(() => {}, 0)` مع تمرير مهلة زمنية مقدارها 0 ميلي ثانية وعن `process.nextTick()`، إذ تُنفَّذ الدالة المُمرَّرة إلى `process.nextTick()` في تكرار حلقة الأحداث الحالي بعد انتهاء العملية الحالية، وهذا يعني أنها ستُنفَّذ دائمًا قبل `setTimeout` و`setImmediate`، كما تشبه دالةُ رد النداء `setTimeout()` مع تأخير 0 ميلي ثانية الدالةَ `setImmediate()`، في حين يعتمد ترتيب التنفيذ على عوامل مختلفة، ولكنهما ستُشغَّلان في التكرار التالي لحلقة الأحداث.

4.2.3 التأخير الصفري Zero delay

إذا حدّدت تأخير المهلة الزمنية بالقيمة 0، فستُنفَّذ دالة رد النداء في أقرب وقت ممكن ولكن بعد تنفيذ الدالة الحالية:

```
setTimeout(() => {
  console.log('after ')
}, 0)
console.log(' before ')
```

ستطبع الدالة السابقة `before after`.

يُعَدّ هذا مفيدًا لتجنب إيقاف وحدة المعالجة المركزية CPU في المهام المكثفة والسماح بتنفيذ الدوال الأخرى أثناء إجراء عملية حسابية ثقيلة عن طريق وضع الدوال ضمن طابور في المجدوِل scheduler.

> تطبّق بعض المتصفحات مثل متصفح IE ومتصفح Edge دالة `setImmediate()` التي تؤدي العملية نفسها، ولكنها ليست معيارًا وغير متوفرة في المتصفحات الأخرى، وإنما هي دالة معيارية في Node.js.

4.2.4 الدالة setInterval()

تُعَدّ `setInterval` دالةً مشابهةً للدالة `setTimeout`، مع اختلاف أنّ الدالة `setInterval` ستشغّل دالةَ رد النداء إلى الأبد ضمن الفاصل الزمني الذي تحدّده مقدَّرًا بالميلي ثانية بدلًا من تشغيلها مرةً واحدةً:

```
setInterval(() => {
 // تشغيل كل 2 ثانية
}, 2000)
```

تُشغَّل الدالة السابقة كل 2 ثانية ما لم تخبرها بالتوقف باستخدام `clearInterval` من خلال تمرير معرِّف `id` الفاصل الزمني الذي تعيده الدالة `setInterval`:

```
const id = setInterval(() => {
 // تشغيل كل 2 ثانية
}, 2000)
clearInterval(id)
```

يشيع استدعاء `clearInterval` ضمن دالة رد نداء الدالة `setInterval`، للسماح لها بالتحديد التلقائي إذا وجب تشغيلها مرةً أخرى أو إيقافها، حيث تشغّل الشيفرة التالية شيئًا على سبيل المثال إذا لم تكن قيمة App.somethingIWait هي `arrived`:

```
const interval = setInterval(() => {
 if (App.somethingIWait === 'arrived') {
   clearInterval(interval)
   return
 }
 // وإلّا افعل شيئًا ما
}, 100)
```

4.2.5 دالة setTimeout العودية

تبدأ `setInterval` دالةً كل n ميلي ثانية، دون الأخذ في الحسبان موعد انتهاء تنفيذ هذه الدالة، فإذا استغرقت الدالة القدر نفسه من الوقت دائمًا، فلا بأس بذلك:

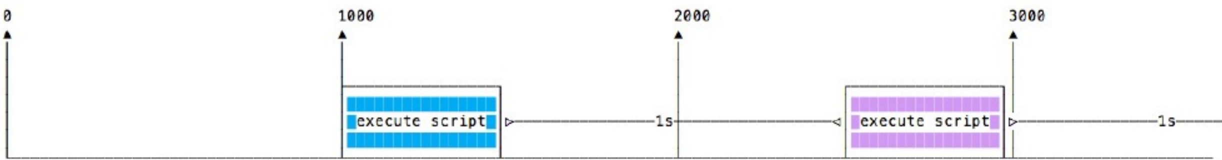
قد تستغرق الدالة أوقات تنفيذ مختلفة اعتمادًا على ظروف الشبكة مثلًا:

وقد يتداخل وقت تنفيذ دالة طويل مع وقت تنفيذ الدالة التالية:

يمكن تجنب ذلك من خلال جدولة دالة `setTimeout` العودية لتُستدعَى عند انتهاء دالة رد النداء:

```
const myFunction = () => {
 // افعل شيئًا ما
 setTimeout(myFunction, 1000)
}
setTimeout(
 myFunction()
}, 1000)
```

بهدف تحقيق السيناريو التالي:



يتوفَّر كل من `setTimeout` و`setInterval` في Node.js من خلال وحدة المؤقتات، كما يوفِّر Node.js أيضًا الدالة `setImmediate()` التي تعادل استخدام `setTimeout(() => {}, 0)` المستخدَمة للعمل مع حلقة أحداث Node.js في أغلب الأحيان.

4.3 مطلق الأحداث Event Emitter الخاص بنود Node

إذا استخدمت جافا سكريبت في المتصفح سابقًا، فلا بد أنك تعرف مقدار تفاعلات المستخدِم المُعالَجة من خلال الأحداث مثل نقرات الفأرة وضغطات أزرار لوحة المفاتيح والتفاعل مع حركة الفأرة وغير ذلك، وهنالك الكثير من الأحداث الأساسية في المتصفح ولكن قد تحتاج في وقت ما إلى أحداث مخصَّصة غير تلك الأساسية لتطلقها وفقًا لوقوع حدث ما ثم تعالجها بما يناسبك.

يوفِّر نود على جانب الواجهة الخلفية خيارًا لإنشاء نظام مماثل باستخدام وحدة الأحداث events module، إذ تقدّم هذه الوحدة الصنف `EventEmitter` الذي يُستخدَم لمعالجة الأحداث، كما يمكنك تهيئته كما يلي:

```
const eventEmitter = require('events').EventEmitter()
```

يُظهِر هذا الكائن التابعين `on` و`emit` من بين أشياء متعددة.

- `emit` الذي يُستخدَم لبدء حدث.
- `on` الذي يُستخدَم لإضافة دالة رد نداء والتي ستُنفَّذ عند بدء الحدث.

لننشئ حدث `start` مثلًا ثم نتفاعل معه من خلال تسجيل الدخول إلى الطرفية:

```
eventEmitter.on('start', () => {
 console.log('started')
})
```

فإذا شغّلنا ما يلي:

```
eventEmitter.emit('start')
```

فستُشغَّل دالة معالج الأحداث، وسنحصل على سجل طرفية.

يمكنك تمرير الوسائط إلى معالج الأحداث من خلال تمريرها على أساس وسائط إضافية إلى التابع `emit()` كما يلي:

```
eventEmitter.on('start', (number) => {
 console.log(`started ${number}`)
})
eventEmitter.emit('start', 23)
```

أو من خلال تمرير وسائط متعددة كما يلي:

```
eventEmitter.on('start', (start, end) => {
```

```
  console.log(`started from ${start} to ${end}`)
})
eventEmitter.emit('start', 1, 100)
```

يظهِر كائن EventEmitter توابعًا متعددةً أخرى للتفاعل مع الأحداث مثل:

- `once()`: يضيف مستمعًا لمرة واحدة.
- `removeListener()` أو `off()`: يزيل مستمع حدث من الحدث.
- `removeAllListeners()`: يزيل جميع المستمعين لحدث ما.

يمكنك قراءة جميع التفاصيل الخاصة بهذه التوابع في صفحة الأحداث في Node.js.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

5. البرمجة غير المتزامنة في Node.js

تُعَدّ الحواسيب غير متزامنة في تصميمها، ويعني المصطلح غير متزامن Asynchronous أنّ الأشياء يمكن حدوثها حدوثًا مستقلًا عن تدفق البرنامج الرئيسي، إذ يُشغَّل كل برنامج لفتحة زمنية محدَّدة في الحواسيب الاستهلاكية الحالية، ثم يتوقف تنفيذه للسماح لبرنامج آخر بمواصلة التنفيذ، حيث تجري هذه العملية ضمن دورة سريعة جدًا بحيث لا يمكن ملاحظتها، وبالتالي نعتقد أن الحواسيب تشغّل عدة برامج في الوقت نفسه، لكن ذلك وهم باستثناء الأجهزة متعددة المعالجات.

تستخدم البرامج المقاطعات interrupts داخليًا، فالمقاطعة هي إشارة تنبعث من المعالج لجذب انتباه النظام، ولن نخوض في التفاصيل الداخلية، ولكن ضع في بالك أن عدم تزامن البرامج أمرٌ طبيعي، إذ توقف تنفيذها إلى أن تتنبّه مرةً أخرى، بحيث يمكن للحاسوب تنفيذ أشياء أخرى في هذه الأثناء، فإذا انتظر برنامج استجابةً من الشبكة، فلا يمكن إيقاف المعالج إلى أن ينتهي الطلب.

تكون لغات البرمجة متزامنةً عادةً، وتوفّر بعضها طريقةً لإدارة عدم التزامن في اللغة نفسها أو من خلال المكتبات، فاللغات C و Java و C# و PHP و Go و Ruby و Swift و Python متزامنة افتراضيًا، كما تعالِج بعضها عدم التزامن باستخدام الخيوط threads، مما ينتج عنه عملية جديدة، فلغة جافا سكريبت متزامنة

افتراضيًا وتعمل على خيط وحيد، وهذا يعني أنّ الشيفرة لا يمكنها إنشاء خيوط جديدة وتشغيلها على التوازي، إذ تُنفَّذ سطور الشيفرة تسلسليًا سطرًا تلو الآخر كما في المثال التالي:

```
const a = 1
const b = 2
const c = a * b
console.log(c)
doSomething()
```

نشأت جافا سكريبت داخل المتصفح، وكانت وظيفتها الرئيسية في البداية الاستجابة لإجراءات المستخدِم مثل onClick وonMouseOver وonChange وonSubmit وما إلى ذلك، ولكن بيئتها ساعدتها في التعامل مع نمط البرمجة المتزامن من خلال المتصفح الذي يوفّر مجموعةً من واجهات برمجة التطبيقات APIs التي يمكنها التعامل مع هذا النوع من العمليات، كما قدّم Node.js في الآونة الأخيرة بيئة إدخال/إخراج دون توقف لتوسيع هذا المفهوم ليشمل الوصول إلى الملفات واستدعاءات الشبكة وغير ذلك.

5.1 دوال رد النداء Callbacks

لا يمكنك معرفة الوقت الذي سينقر فيه المستخدِم على زر، لذلك تعرِّف معالج أحداث لحدث النقر الذي يقبل دالةً تُستدعَى عند بدء الحدث كما يلي:

```
document.getElementById('button').addEventListener('click', () => {
  //نُقِر العنصر
})
```

وهذا ما يسمى دالة رد النداء، وهي دالة بسيطة تُمرَّر على أساس قيمة إلى دالة أخرى وستُنفَّذ عند وقوع الحدث فقط، إذ يمكن ذلك لأن للغة جافا سكريبت دوالًا من الصنف الأول، والتي يمكن إسنادها للمتغيرات وتمريرها إلى دوال أخرى تسمّى دوال الترتيب الأعلى higher-order functions، كما تُغلَّف شيفرة العميل في مستمع حدث load على الكائن window الذي يشغِّل دالة رد النداء عندما تكون الصفحة جاهزة فقط مثل المثال التالي:

```
window.addEventListener('load', () => {
  //حُمِّلت الصفحة
  //افعل ما تريده
})
```

تُستخدَم دوال رد النداء في كل مكان، ولا تقتصر على أحداث DOM فقط، فأحد الأمثلة الشائعة عليها هو استخدام المؤقتات:

```
setTimeout(() => {
  // تشغيل بعد 2 ثانية
}, 2000)
```

تقبَل طلبات XHR دالة رد نداء عن طريق إسناد دالة لخاصية في المثال التـالي، إذ ستُسـتدعَى هـذه الدالـة عند وقوع حدث معيّن -أي حدث تغيّرات حالة الطلب في مثالنا-:

```
const xhr = new XMLHttpRequest()
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    xhr.status === 200 ? console.log(xhr.responseText) : 
console.error('error')
  }
}
xhr.open('GET', 'https://yoursite.com')
xhr.send()
```

5.1.1 معالجة الأخطاء في دوال رد النداء

تتمثَّل إحدى الإستراتيجيات الشائعة جدًا في استخدام ما يعتمده Node.js وهو المعامل الأول في أيّ دالـة رد نداء هي كائن الخطأ، وبالتالي تُسمّى دوال رد النداء مع معامل الأخطاء الأول error-first callbacks، فإذا لم يكن هنـاك خطـأً، فسـتكون قيمـة الكـائن null، وإذا كـان هنـاك خطـأ، فسـيحتوي هـذا الكـائن وصـفًا للخطـأ ومعلومات أخرى.

```
fs.readFile('/file.json', (err, data) => {
  if (err !== null) {
    //عالِج الخطأ
    console.log(err)
    return
  }
  //لا يوجد خطأ، إذًا عالِج البيانات
  console.log(data)
})
```

5.1.2 مشكلة دوال رد النداء

تُعَـدّ دوال رد النـداء رائعـةً في الحـالات البسـيطة، ولكن تضـيف كـل دالـة رد نـداء مسـتوىً من التـداخل nesting، وبالتالي تتعقَّد الشيفرة بسرعة كبيرة عند وجود كثير من دوال رد النداء كما يلي:

```
window.addEventListener('load', () => {
  document.getElementById('button').addEventListener('click', () => {
    setTimeout(() => {
      items.forEach(item => {
        //أضِف شيفرتك هنا
      })
    }, 2000)
  })
})
```

تُعَدّ الشيفرة السابقة بسيطةً، إذ تتألف من 4 مستويات فقط، لكنك قد تصادف مستويات أكثر بكثير من التداخل وبالتالي سيزداد تعقيد الشيفرة.

5.1.3 بدائل دوال رد النداء

قدّمت جافا سكريبت بدءًا من الإصدار ES6 ميزات متعددةً تساعدنا في التعامل مع الشيفرة غير المتزامنة التي لا تتضمن استخدام دوال رد النداء مثل:

- الوعود Promises في الإصدار ES6.
- صيغة عدم التزامن/الانتظار Async/Await في الإصدار ES8.

5.2 الوعود Promises

الوعود هي إحدى طرق التعامل مع الشيفرات غير المتزامنة في جافا سكريبت دون كتابة كثير من دوال رد النداء في الشيفرة.

5.2.1 مدخل إلى الوعود

يُعرَّف الوعد Promise عمومًا على أنه وكيل لقيمة ستتوفر في وقت لاحق، فالوعود موجودة منذ سنوات، لكن وُحِّدت وقُدِّمت في الإصدار ES2015، واُستبدِلت دوال عدم التزامن Async functions في الإصدار ES2017 بها والتي تستخدِم واجهة برمجة تطبيقات الوعود أساسًا لها، لذلك يُعَدّ فهم الوعود أمرًا أساسيًا حتى في حالة استخدام دوال عدم التزامن في الشيفرة الأحدث عوضًا عن الوعود، وإليك شرحًا مختصرًا عن كيفية عمل الوعود.

يبدأ الوعد عند استدعائه في حالة انتظار pending state، أي أن الدالة المستدعِية تواصل التنفيذ في الوقت الذي تنتظر به الوعد لينفّذ معالجته الخاصة ويستجيب لها، حيث تنتظر الدالة المستدعِية إما إعادة الوعد في حالة التأكيد أو الحل resolved state أو في حالة الرفض rejected state، ولكن لغة جافا سكريبت غير

متزامنة، لذلك تتابع الدالة تنفيذها ريثما ينتهي الوعد من عمله، كما تستخدِم واجهات برمجة تطبيقات الويب المعيارية الحديثة الوعود بالإضافة إلى شيفرتك ومكتباتها، ومن هذه الواجهات البرمجية:

- Battery API
- Fetch API
- Service Workers

ستستخدَم الوعود بالتأكيد في جافا سكريبت الحديثة، لذلك يجب فهمها جيدًا.

5.2.2 إنشاء وعد

تُظهِر واجهة برمجة الوعد Promise API باني وعد Promise constructor يمكن تهيئته باستخدام الدالة `new Promise()`:

```
let done = true
const isItDoneYet = new Promise(
  (resolve, reject) => {
    if (done) {
      const workDone = 'Here is the thing I built'
      resolve(workDone)
    } else {
      const why = 'Still working on something else'
      reject(why)
    }
  }
)
```

يتحقّق الوعد من الثابت العام `done`، فإذا كانت قيمته صحيحة `true`، فإننا نعيد قيمة وعد مؤكَّد، وإلا فسنعيد وعدًا مرفوضًا، كما يمكننا إعادة قيمة باستخدام القيم `resolve` و`reject`، حيث أعدنا سلسلةً نصيةً فقط في المثال السابق، لكنها يمكن أن تكون كائنًا أيضًا.

5.2.3 استهلاك وعد

لنرى الآن كيفية استهلاك أو استخدام وعد.

```
const isItDoneYet = new Promise(
  //...
)
```

```
const checkIfItsDone = () => {
  isItDoneYet
    .then((ok) => {
      console.log(ok)
    })
    .catch((err) => {
      console.error(err)
    })
}
```

سيؤدي تشغيل الدالة `checkIfItsDone()` إلى تنفيذ الوعد `isItDoneYet()` وسـتنتظر إلى أن يُؤكَّـد الوعد باستخدام دالة رد النداء `then`، وإذا كان هناك خطأ، فستعالجه في دالة رد النداء `catch`.

إذا أردت مزامنة وعود مختلفـة، فسيسـاعدك التـابع `Promise.all()` على تحديـد قائمـة وعـود، وتنفيـذ شيء ما عند تأكيد هذه الوعود جميعها، وإليك المثال التالي:

```
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')
Promise.all([f1, f2]).then((res) => {
  console.log('Array of results', res)
})
.catch((err) => {
  console.error(err)
})
```

تتيح لك صيغة إسناد الهـدم destructuring assignment syntax الخاصـة بالإصـدار ES2015 تنفيـذ ما يلي:

```
Promise.all([f1, f2]).then(([res1, res2]) => {
  console.log('Results', res1, res2)
})
```

ليس الأمـر مقتصـرًا على اسـتخدام `fetch` بـالطبع، إذ يمكنـك اسـتخدام أيّ وعـد، كمـا يُشـغِّل التـابع `Promise.race()` عند تأكيد أول وعد من الوعود التي تمرّرها إليه، ويشغِّل دالةَ رد النداء المصاحبة للوعد مـرةً واحدةً فقط مع نتيجة الوعد الأول المُؤكَّد resolved، وإليك المثال التالي:

```
const first = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'first')
```

```
})
const second = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'second')
})
Promise.race([first, second]).then((result) => {
  console.log(result) // second
})
```

5.2.4 سلسلة الوعود Chaining promises

يمكن أن يُعاد وعدٌ إلى وعد آخر، وبالتالي ستنشأ سلسلة من الوعود، إذ تقدِّم واجهة Fetch API -وهي طبقة فوق واجهة برمجة تطبيقات XMLHttpRequest- مثالًا جيدًا عن سلسلة وعود، إذ يمكننا استخدام هذه الواجهة للحصول على مورد ووضع سلسلة من الوعود في طابور لتنفيذها عند جلب المورد.

كما تُعَدّ واجهة Fetch API آليةً قائمةً على الوعود، حيث يكافئ استدعاءُ الدالة `fetch()` تعريف وعد باستخدام `new Promise()`، وإليك المثال التالي عن كيفية سلسلة الوعود:

```
const status = (response) => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  }
  return Promise.reject(new Error(response.statusText))
}
const json = (response) => response.json()

fetch('/todos.json')
  .then(status)
  .then(json)
  .then((data) => { console.log('Request succeeded with JSON
response', data) })
  .catch((error) => { console.log('Request failed', error) })
```

نستدعي في المثال السابق التابع `fetch()` للحصول على قائمة من عناصر TODO من ملف `todos.json` الموجود في نطاق الجذر، وننشئ سلسلة من الوعود. كما يعيد تشغيل التابع `fetch()` استجابةً لها خاصيات منها:

- `status` وهي قيمة عددية تمثِّل رمز حالة HTTP.
- `statusText` وهي رسالة حالة تكون قيمتها OK إذا نجح الطلب.

تحتوي الاستجابة response أيضًا على تابع json() الذي يعيد وعدًا سيُؤكد ويُربَط مع محتوى الجسم المُعالَج والمُحوَّل إلى JSON.

الوعد الأول في السلسلة هو الدالة التي حدّدناها وهي status() التي تتحقق من حالة الاستجابة، فإذا لم تكن استجابةً ناجحةً -أي قيمتها بين 200 و299، فسترفض الوعد، إذ ستؤدي هذه العملية إلى تخطي جميع الوعود المتسلسلة المدرجَة في سلسلة الوعود وستنتقل مباشرةً إلى تعليمة catch() في الأسفل، مما يؤدي إلى تسجيل نص فشل الطلب Request failed مع رسالة الخطأ.

أما إذا نجحت الاستجابة، فستُستدعَى دالة json() التي حدّدناها، وبما أنّ الوعد السابق يعيد كائن الاستجابة response عند النجاح، فسنحصل عليه على أساس دخل للوعد الثاني، وبالتالي نُعيد بيانات JSON المُعالَجة في هذه الحالة، لذا فإن الوعد الثالث يتلقى JSON مباشرةً مع تسجيله ببساطة في الطرفية كما يلي:

```
.then((data) => {
  console.log('Request succeeded with JSON response', data)
})
```

5.2.5 معالجة الأخطاء

ألحقنا في المثال السابق تعليمة catch بسلسلة وعود، فإذا فشل أيّ شيء في سلسلة الوعود مسببًا خطأً أو رفض وعد، فسينتقل التحكم إلى أقرب تعلمية catch() أسفل السلسلة.

```
new Promise((resolve, reject) => {
  throw new Error('Error')
})
  .catch((err) => { console.error(err) })
// أو
new Promise((resolve, reject) => {
  reject('Error')
})
  .catch((err) => { console.error(err) })
```

إذا ظهر خطأ ضمن تعليمة catch()، فيمكنك إلحاق تعليمة catch() ثانية لمعالجة الخطأ وهلمَّ جرًا وهذا ما يسمى بعملية معالجة توريث الأخطاء Cascading errors.

```
new Promise((resolve, reject) => {
  throw new Error('Error')
})
  .catch((err) => { throw new Error('Error') })
```

```
    .catch((err) => { console.error(err) })
```

إذا ظهر الخطأ `Uncaught TypeError: undefined is not a promise` في الطرفية، فتأكد من استخدام `new Promise()` بدلًا من استخدام `Promise()`.

5.3 صيغة عدم التزامن أو الانتظار async/await

تطوّرت لغة جافا سكريبت في وقت قصير جدًا من دوال رد النداء callbacks إلى الوعود promises في الإصدار ES2015، وأصبحت لغة جافا سكريبت غير المتزامنة منذ الإصدار ES2017 أبسط مع صيغة عدم التزامن أو الانتظار async/await، فالدوال غير المتزامنة هي مزيج من الوعود والمولِّدات generators، وهي في الأساس ذات مستوىً أعلى من الوعود من ناحية التجريد، فصيغة async/await مبنية على الوعود.

سبب ظهور صيغة async/await هو أنها تقلل من الشيفرة التكرارية أو المتداولة boilerplate الموجودة في الوعود، وتقلّل من محدودية قيود عدم كسر السلسلة في تسلسل الوعود، فقد كان الهدف من تقديم الوعود في الإصدار ES2015 حلَّ مشكلة التعامل مع الشيفرة غير المتزامنة، وقد حلّت هذه المشكلة حقًا، ولكن كان واضحًا على مدار العامين اللذين فصلا بين الإصدارين ES2015 وES2017 أن الوعود ليست الحل النهائي.

أُستخدِمت الوعود لحل مشكلة تعرف باسم جحيم دوال رد النداء callback hell الشهيرة، لكنها أدخلت التعقيد فيها بالإضافة إلى تعقيد الصيغ، وقد كانت عناصر أولية جيدة يمكن من خلالها إظهار صيغة أفضل للمطورين، لذلك حصلنا على دوال غير متزامنة في الوقت المناسب، إذ تظهر الشيفرة على أنها متزامنة، لكنها غير متزامنة وغير قابلة للتوقّف non-blocking في الحقيقة.

5.3.1 كيفية عمل صيغة async/await

تعيد الدالة غير المتزامنة وعدًا كما في المثال التالي:

```
const doSomethingAsync = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
```

إذا أردت استدعاء هذه الدالة، فستضيف الكلمة `await` في البداية، وستتوقف شيفرة الاستدعاء حتى تأكيد أو رفض الوعد.

> يجب تعريف دالة العميل على أنها غير متزامنة `async`.

إليك المثال التالي:

```
const doSomething = async () => {
  console.log(await doSomethingAsync())
}
```

إليك المثال التالي أيضًا والذي يوضِّح استخدام صيغة async/await لتشغيل دالة تشغيلًا غير متزامن:

```
const doSomethingAsync = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
const doSomething = async () => {
  console.log(await doSomethingAsync())
}
console.log('Before')
doSomething()
console.log('After')
```

ستطبع الشيفرة السابقة ما يلي في طرفية المتصفح:

```
Before
After
I did something //after 3s
```

5.3.2 تطبيق الوعود على كل شيء

تعني إضافة الكلمة المفتاحية async في بداية أيّ دالة أنّ هذه الدالة ستعيد وعدًا، وإذا لم تفعل ذلك صراحةً، فستعيد وعدًا داخليًا.

هذا سبب كون الشيفرة التالية صالحةً valid:

```
const aFunction = async () => {
  return 'test'
}
aFunction().then(alert) // سيؤدي هذا إلى تنبيه 'test'
```

وكذلك الشيفرة التالية:

```
const aFunction = async () => {
  return Promise.resolve('test')
}
aFunction().then(alert) // 'test' سيؤدي هذا إلى تنبيه
```

تبدو الشيفرة السابقة بسيطةً للغاية إذا وازنتها مع الشيفرة التي تستخدِم وعودًا صريحةً مع الدوال المتسلسلة ودوال رد النداء، كما يُعَدّ المثال السابق بسيطًا للغاية، لذلك ستظهر الفوائد جليةً عندما تكون الشيفرة أكثر تعقيدًا، وإليك المثال التالي الذي يوضّح كيفية الحصول على مورد JSON وتحليله parse باستخدام الوعود:

```
const getFirstUserData = () => {
  return fetch('/users.json') // الحصول على قائمة المستخدِمين
    .then(response => response.json()) // JSON تحليل
    .then(users => users[0]) // التقاط المستخدِم الأول
    .then(user => fetch(`/users/${user.name}`)) //  الحصول على بيانات المستخدِم
    .then(userResponse => response.json()) // JSON تحليل
}
getFirstUserData()
```

وإليك المثال التالي الذي ينفّذ ما يفعله المثال السابق ولكن باستخدام صيغة await/async:

```
const getFirstUserData = async () => {
  const response = await fetch('/users.json') //  الحصول على قائمة المستخدِمين
  const users = await response.json() // JSON تحليل
  const user = users[0] // التقاط المستخدِم الأول
  const userResponse = await fetch(`/users/${user.name}`) // الحصول على بيانات المستخدِم
  const userData = await user.json() // JSON تحليل
  return userData
}
getFirstUserData()
```

5.3.3 استخدام دوال متعددة غير متزامنة ضمن سلسلة

يمكن وضع الدوال غير المتزامنة ضمن سلسلة بسهولة باستخدام صيغة أكثر قابلية للقراءة من الوعود الصرفة كما يلي:

```
const promiseToDoSomething = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 10000)
  })
}
const watchOverSomeoneDoingSomething = async () => {
  const something = await promiseToDoSomething()
  return something + ' and I watched'
}
const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {
  const something = await watchOverSomeoneDoingSomething()
  return something + ' and I watched as well'
}
watchOverSomeoneWatchingSomeoneDoingSomething().then((res) => {
  console.log(res)
})
```

ستطبع الشيفرة السابقة ما يلي:

```
I did something and I watched and I watched as well
```

5.3.4 سهولة تنقيح الأخطاء

يُعَدّ تنقيح أخطاء Debugging الوعود أمرًا صعبًا لأن منقِّح الأخطاء لن يتخطى الشيفرة غير المتزامنة، بينما تجعل صيغة Async/await هذا الأمر سهلًا لأنها تُعَدّ مجرد شيفرة متزامنة بالنسبة للمصرِّف compiler.

دورة إدارة تطوير المنتجات

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

6. التعامل مع طلبيات الشبكية في Node.js

نتعرّف من خلال هذا الفصل على طريقة إرسال واستقبال الطلبيـات بين الخـادم والعميـل عـبر الشـبكة في Node.js باستخدام مكتبة Axios، سنبدأ بشرح وسيلة التواصل الأساسية بين الخادم والمتصفح وهو بروتوكـول HTTP، وسنتعرّف على بديل اتصال HTTP في تطبيقات الويب الذي هو مقابس الويب WebSockets.

6.1 كيفية عمل بروتوكول HTTP

يُعَدّ بروتوكول نقل النص الفائق Hyper Text Transfer Protocol -أو HTTP اختصارًا- أحد بروتوكـولات تطبيق TCP/IP وهي مجموعة البروتوكولات التي تشغّل شبكة الإنترنت، إذ يُعَدّ البروتوكول الأنجح والأكثر شعبية على الإطلاق، كما يُشغِّل هذا البروتوكول شبكة الويب العالمية World Wide Web، مما يمنح المتصفحات لغة للتواصل مع الخوادم البعيدة التي تستضيف صفحات الويب.

وُحِّد بروتوكول HTTP لأول مرة في عام 1991على أساس نتيجـة لعمـل تيم بـيرنرز لي Tim Berners-Lee في المركز الأوروبي للأبحاث النووية European Center of Nuclear Research -أو CERN اختصارًا- منذ عام 1989، وكان الهدف هو السماح للباحثين بتبادل أبحـاثهم بسـهولة وربطهم ببعضـهم بعضًـا على أسـاس وسـيلة تحسّن عمل المجتمع العلمي، كما تكوّنت تطبيقات الإنترنت الرئيسـية في ذلـك الـوقت من بروتوكـول FTP أي بروتوكول نقل الملفات File Transfer Protocol والبريـد الإلكـتروني ونظـام يـوزنت Usenet أي مجموعـات الأخبار newsgroups، ولكنها أصبحت غير مُستخدَمة حاليًا تقريبًا.

صدر متصفح موزاييك Mosaic في عام 1993، وهو أول متصفح ويب رسومي، وتطورت الأمور عنــدها، إذ أصبح الويب التطبيق الرائج في شبكة الإنـترنت، حيث سـبّب ظهـوره ضـجةً كبـيرةً، كمـا تطـوّر الـويب والنظـام المجتمعي المحيط به تطوّرًا كبيرًا بمرور الوقت مع بقاء الأساسيات على حالها، وأحد الأمثلة على هذا التطور هــو

أنّ بروتوكول HTTP يشغّل حاليًا -بالإضافة إلى صفحات الويب- واجهات برمجة تطبيقات REST، وهي إحدى الطرق الشائعة للوصول إلى خدمة عبر الإنترنت برمجيًا.

عُدِّل بروتوكول HTTP تعديلًا ثانويًا في عام 1997 في الإصدار HTTP/1.1، وخلفه الإصدار HTTP/2 الذي وُحِّد في عام 2015 ويُطبَّق الآن على خوادم الويب الرئيسية المُستخدَمة في جميع أنحاء العالم، كما يُعَدّ بروتوكول HTTP غير آمن مثل أيّ بروتوكول آخر غير مخدَّم عبر اتصال مشفَّر مثل بروتوكولات SMTP وFTP وغيرها، وهذا هو السبب في التوجّه الكبير حاليًا نحو استخدام بروتوكول HTTPS، وهو بروتوكول HTTP مخدَّم عبر بروتوكول TLS، ولكن بروتوكول HTTP هو حجر الأساس لبروتوكول HTTP/2 وHTTPS.

6.1.1 مستندات HTML

بروتوكول HTTP هو الطريقة التي تتواصل بها متصفحات الويب web browsers مثل Chrome و Firefox وEdge ومتصفحات أخرى سنسمّيها عملاء clients مع خوادم الويب web servers، كما اُشتق الاسم بروتوكول نقل النص الفائق Hyper Text Transfer Protocol من الحاجة إلى نقل الملفات كما هو الحال في بروتوكول FTP والذي يشير إلى بروتوكول نقل الملفات File Transfer Protocol، بالإضافة إلى النصوص الفائقة hypertexts التي ستُكتَب باستخدام لغة HTML، ثم تُمثَّل رسوميًا باستخدام المتصفح مع عرض جميل وروابط تفاعلية، وساهمت الروابط بقوة في اعتماد بروتوكول HTTP إلى جانب سهولة إنشاء صفحات ويب جديدة، حيث ينقل هذا البروتوكول ملفات النصوص الفائقة بالإضافة إلى الصور وأنواع الملفات الأخرى عبر الشبكة.

6.1.2 الروابط والطلبيات

يمكن أن يؤشّر مستند إلى مستند آخر باستخدام الروابط ضمن متصفح الويب، حيث يحدِّد جزء الرابط الأول كلًا من البروتوكول وعنوان الخادم من خلال إما اسم نطاق domain name أو عنوان IP، وليس هذا الجزء خاصًا ببروتوكول HTTP؛ أما الجزء الثاني فهو جزء المستند الذي يتبع جزء العنوان ويمثِّل مسار المستند مثل `https://flaviocopes.com/http/` الذي يتكوّن مما يلي:

- `https` هو البروتوكول.
- `flaviocopes.com` هو اسم النطاق الذي يؤشر إلى الخادم.
- `/http/` هو عنوان URL النسبي للمستند إلى مسار الخادم الجذر.

يمكن أن يتداخل المسار مثل `https://academy.hsoub.com/files/c5-programming/`، حيث يكون عنوان URL للمستند هو `/files/c5-programming/`؛ أما خادم الويب فيُعَدّ مسؤولًا عن تفسير الطلب وتقديم الاستجابة الصحيحة بعد تحليل الطلب، كما يمكن أن يكون الطلب عنوان URL الذي رأيناه

سابقًا، فإذا أدخلنا عنوانًا وضغطنا `Enter` من لوحة المفاتيح في المتصفح، فسيرسل الخادم طلبًا في الخلفية إلى عنوان IP الصحيح مثل الطلب التالي:

```
GET /a-page
```

حيث `/a-page` هو عنوان URL الذي طلبته، كما يمكن أن يكون الطلب تابع HTTP ويُسمّى فعلًا verb أيضًا، حيث حدّد بروتوكول HTTP سابقًا ثلاثةً من هذه التوابع وهي:

- `GET`
- `POST`
- `HEAD`

وقدّم الإصدار HTTP/1.1 التوابع:

- `PUT`
- `DELETE`
- `OPTIONS`
- `TRACE`

والتي سنتحدّث عنها لاحقًا، وقد يكون الطلب مجموعة ترويسات HTTP، فالترويسات Headers هي مجموعة من أزواج مفتاح وقيمة `key:value` تُستخدَم للتواصل مع المعلومات الخاصة بالخادم المُحدَّدة مسبقًا ليتمكّن الخادم من فهم ما نعنيه، كما أنّ جميع الترويسات اختيارية باستثناء الترويسة `Host`.

6.1.3 توابع HTTP

أهم توابع HTTP هي:

- `GET`: هو التابع الأكثر استخدامًا، وهو الخيار الذي يُستخدَم عند كتابة عنوان URL في شريط عنوان المتصفح، أو عند النقر على رابط، كما يطلب هذا التابع من الخادم إرسال المورد المطلوب على أساس استجابة.
- `HEAD`: يتشابه هذا التابع مع التابع `GET` تمامًا، ولكن `HEAD` يخبر الخادم بعدم إرسال جسم الاستجابة response body، بل إرسال الترويسات فقط.
- `POST`: يستخدِم العميل هذا التابع لإرسال البيانات إلى الخادم، حيث يُستخدَم عادةً في النماذج forms مثلًا، وعند التفاعل مع واجهة برمجة تطبيقات REST.
- `PUT`: يهدف هذا التابع إلى إنشاء مورد في عنوان URL المحدَّد باستخدام المعاملات المُمرَّرة في جسم الطلب، كما يُستخدم استخدامًا رئيسيًا في واجهات برمجة تطبيقات REST.

- `DELETE`: يُستدعَى هذا التابع مع عنوان URL لطلب حذف المورد المقابل لهذا العنوان، كما يُستخدَم استخدامًا رئيسيًا في واجهات برمجة تطبيقات REST.
- `OPTIONS`: يجب أن يرسِل الخادم قائمة توابع HTTP المسموح بها إلى عنوان URL المحدَّد عندما يتلقى طلب OPTIONS.
- `TRACE`: يعيد هذا التابع إلى العميل الطلب المُستلَم، حيث يُستخدَم هذا التابع لتنقيح الأخطاء debugging أو لأغراض التشخيص.

6.1.4 اتصال HTTP خادم/عميل

بروتوكول HTTP هو بروتوكول عديم الحالة stateless مثل معظم البروتوكولات التي تنتمي إلى مجموعة بروتوكولات TCP/IP، إذ ليس لدى الخوادم أيّ فكرة عن حالة العميل الحالية، فكل ما يهم الخوادم هو أن تتلقى طلبات ثم تعمل على تلبيتها، كما لا يكون لطلب مسبق أيّ معنى في هذا السياق، وبالتالي يمكن أن يكون خادم الويب سريعًا جدًا، مع وجود قليل من المعالجة وحيز نطاق تراسلي bandwidth مناسب لمعالجة كثير من الطلبات المتزامنة.

يُعَدّ بروتوكول HTTP مرنًا واتصاله سريعًا جدًا اعتمادًا على حِمل الشبكة، وهذا يتناقض مع البروتوكولات الأكثر استخدامًا في وقت صدوره مثل TCP وPOP/SMTP وبروتوكولات البريد التي تتضمن كثيرًا من عمليات المصافحة handshaking والتأكيدات على النهايات المُستقبَلة، كما تجرِّد المتصفحات الرسومية هذا الاتصال، ولكن يمكن توضيحه كما يلي، إذ يبدأ سطر الرسالة الأول بتابع HTTP ثم مسار المَورد النسبي وإصدار البروتوكول كما يلي:

```
GET /a-page HTTP/1.1
```

ثم يجب إضافة ترويسات طلبات HTTP، إذ توجد هناك ترويسات متعددة، ولكن الترويسة الإلزامية الوحيدة هي `Host`:

```
GET /a-page HTTP/1.1
Host: flaviocopes.com
```

يمكنك اختبار ذلك باستخدام أداة telnet، وهي أداة سطر أوامر تتيح لنا الاتصال بأي خادم وإرسال الأوامر إليه، والآن افتح طرفيتك terminal واكتب `telnet flaviocopes.com 80` مثلًا، حيث سيؤدي ذلك إلى فتح طرفية تعرض ما يلي:

```
Trying 178.128.202.129...
Connected to flaviocopes.com.
Escape character is '^]'.
```

أنت الآن متصل بخادم الويب Netlify، ثم اكتب ما يلي:

```
GET /axios/ HTTP/1.1
Host: flaviocopes.com
```

اضغط بعد ذلك على زر `Enter` في سطر فارغ لتشغيل الطلب، وستكون الاستجابة كما يلي:

```
HTTP/1.1 301 Moved Permanently
Cache-Control: public, max-age=0, must-revalidate
Content-Length: 46
Content-Type: text/plain
Date: Sun, 29 Jul 2018 14:07:07 GMT
Location: https://flaviocopes.com/axios/
Age: 0
Connection: keep-alive
Server: Netlify
Redirecting to https://flaviocopes.com/axios/
```

وهذه هي استجابة HTTP التي حصلنا عليها من الخـادم، وهي طلب 301 Moved Permanently الـذي يخبرنا بانتقال المَورد إلى موقع آخر انتقالًا دائمًا، وذلك لأننا اتصلنا بالمنفذ 80 وهو المنفذ الافتراضي لـبروتوكول HTTP، ولكننا ضبطنا الخادم على إعـادة التوجيـه التلقـائي إلى HTTPS، كمـا حُـدِّد الموقـع الجديـد في ترويسـة استجابة HTTP التي هي `Location`، وهناك ترويسات استجابة أخرى سنتحدث عنها لاحقًا، كما يفصـل سـطر فـارغ ترويسـة الطلب عن جسـمه في كـل من الطلب والاسـتجابة، حيث يحتـوي جسـم الطلب في مثالنـا على السلسلة النصية التالية:

```
Redirecting to https://flaviocopes.com/axios/
```

يبلغ طول هذه السلسلة النصية 46 بايتًـا كمـا هـو محـدَّد في ترويسـة `Content-Length`، إذ تظهـر هـذه السلسلة في المتصفح عند فتح الصفحة ريثمـا يُعـاد توجيهـك إلى الموقـع الصـحيح تلقائيًـا، كمـا نسـتخدم أداة telnet في مثالنا، وهي أداة منخفضة المستوى يمكننا استخدامها للاتصال بأي خادم، لذلك لا يمكننـا الحصـول على أي نوع من إعادة التوجيه التلقائي، فلنتصل الآن بالمنفذ 443 وهو المنفـذ الافتراضـي لـبروتوكول HTTPS، حيث لا يمكننا استخدام أداة telnet بسـبب مصـافحة SSL الـتي يجب أن تحـدث،فولنسـتخدم الآن أداة `curl` وهي أداة سطر أوامر أخرى، إذ لا يمكننا كتابة طلب HTTP مباشرةً، لكننا سنرى الاستجابة:

```
curl -i https://flaviocopes.com/axios/
```

سنحصل في المقابل على ما يلي:

```
HTTP/1.1 200 OK
Cache-Control: public, max-age=0, must-revalidate
Content-Type: text/html; charset=UTF-8
Date: Sun, 29 Jul 2018 14:20:45 GMT
Etag: "de3153d6eacef2299964de09db154b32-ssl"
Strict-Transport-Security: max-age=31536000
Age: 152
Content-Length: 9797
Connection: keep-alive
Server: Netlify

<!DOCTYPE html>
<html prefix="og: http://ogp.me/ns#" lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<title>HTTP requests using Axios</title>
....
```

لن ينقل خادم HTTP ملفات HTML فقط، وإنما يمكنه نقل ملفات أخرى مثل ملفات CSS و JS و SVG و PNG و JPG وأنواع ملفات متعددة أخرى، إذ يعتمد ذلك على الإعداد، فبروتوكول HTTP قادر تمامًا على نقل هذه الملفات، وسيعرف العميل نوع الملف، وبالتالي سيفسرها بالطريقة الصحيحة، وهذه هي الطريقة التي يعمل بها الويب عند استرداد صفحة HTML بواسطة المتصفح، إذ تُفسَّر هذه الصفحة وأي مَورد آخر يحتاجه المتصفح لعرض خاصية (CSS وJavaScript والصور وغير ذلك) مُستَرَدّة عبر طلبات HTTP إضافية إلى الخادم نفسه.

6.1.5 بروتوكول HTTPS والاتصالات الآمنة

يُعَدّ بروتوكول HTTPS امتدادًا لبروتوكول HTTP -أي بروتوكول نقل النص الفائق- والذي يوفّر اتصالًا آمنًا، فبروتوكول HTTP غير آمن في تصميمه، فإذا فتحتَ متصفحك وطلبتَ من خادم الويب إرسال صفحة ويب لك، فستسير بياناتك ضمن رحلتين تكون الأولى من المتصفح إلى خادم الويب، والأخرى من خادم الويب إلى المتصفح، وقد تحتاج بعد ذلك إلى مزيد من الاتصالات -اعتمادًا على محتوى صفحة الويب- للحصول على ملفات CSS وملفات JavaScript والصور وما إلى ذلك، كما يمكن فحص بياناتك والتلاعب بها خلال مرورها في الشبكة أثناء أيّ من هذه الاتصالات.

قد تكون العواقب وخيمةً، فقد يراقب ويسجّل طرف ثالث كل أنشطة شبكتك دون علمك، وقد تحقن بعض الشبكات إعلانات، وقد تكون عرضةً لهجوم الوسيط man-in-the-middle، وهو تهديد أمني يستطيع المهاجم من خلاله التلاعب ببياناتك وحتى انتحال شخصية حاسوبك عبر الشبكة، إذ يمكن لأي شخص الاستماع بسهولة إلى حزم HTTP المُرسَلة عبر شبكة واي فاي Wi-Fi عامة وغير مشفَّرة، حيث يهدف بروتوكول HTTPS إلى حل هذه المشكلة من خلال تشفير الاتصال الكامل بين متصفحك وخادم الويب.

تُعَدّ كل الخصوصية والأمن مصدر قلق كبير في شبكة الإنترنت حاليًا، فقد كان الأمر مختلفًا قبل بضع سنوات، حيث كان بإمكانك توفير الأمن من خلال استخدام اتصال مشفر فقط في الصفحات المحمية بتسجيل الدخول أو أثناء عمليات الدفع في المتاجر الإلكترونية، كما أنّ معظم مواقع الويب قد استخدَمت بروتوكول HTTP بسبب أسعار شهادات SSL وتعقيداتها.

يُعَدّ استخدام HTTPS إلزاميًا على جميع المواقع في الوقت الحالي، إذ يستخدِمه حاليًا أكثر من 50% من مواقع الويب، وقد بدأ Google Chrome مؤخرًا في تمييز مواقع HTTP بأنها غير آمنة، لمنحك سببًا وجيهًا في جعل بروتوكول HTTPS إلزاميًا على جميع مواقع الويب الخاصة بك.

يكون منفذ الخادم الافتراضي هو 80 عند استخدام بروتوكول HTTP، في حين يكون 443 عند استخدام بروتوكول HTTPS، وليست إضافته بصورة صريحة أمرًا إلزاميًا إذا استخدَم الخادم المنفذ الافتراضي، كما يُطلق على بروتوكول HTTPS أحيانًا اسم HTTP عبر SSL أو HTTP عبر TLS، حيث يكون بروتوكول TLS خلَفًا لبروتوكول SSL؛ أما الشيء الوحيد غير المشفَّر عند استخدام بروتوكول HTTPS، فهو نطاق خادم الويب ومنفذ الخادم، بينما تُشفَّر كل المعلومات الأخرى بما في ذلك مسار المَورد والترويسات وملفات تعريف الارتباط cookies ومعامِلات الاستعلام.

لن نشرح تفاصيل تحليل كيفية عمل بروتوكول TLS الداخلي، لكنك قد تعتقد أنه يضيف قدرًا كبيرًا من الحِمل على الشبكة، وربما هذا صحيح، إذ تتسبب أيّ عملية حسابية مُضافَة إلى معالجة موارد الشبكة في زيادة الحِمل على العميل والخادم وحجم الرُزم المرسَلة على حد سواء.

يتيح HTTPS استخدام أحدث بروتوكول وهو HTTP/2 الذي يحتوي على ميزة إضافية يتفوق بها على الإصدار HTTP/1.1، وهذه الميزة هي أنه أسر ع لعدة أسباب مثل ضغط الترويسة وتعدُّد الموارد، كما يمكن للخادم زجّ مزيد من الموارد عند طلب أحدها، فإذا طلب المتصفح صفحةً، فسيتلقى جميع الموارد اللازمة مثل الصور وملفات CSS وJS، كما يُعَدّ HTTP/2 تحسّنًا كبيرًا على HTTP/1.1 ويتطلب بروتوكول HTTPS، وهذا يعني أنّ HTTPS أسر ع من HTTP بكثير إذا ضُبِط كل شيء ضبطًا صحيحًا باستخدام إعداد حديث على الرغم من وجود عبء التشفير الإضافي.

6.2 كيفية عمل طلبات HTTP

سنشرح ما يحدث عند كتابة عنوان URL في المتصفح من البداية إلى النهاية، حيث سنوضّح كيف تطبّق المتصفحات طلبات الصفحة باستخدام بروتوكول HTTP/1.1، إذ ذكرنا HTTP على وجه الخصوص لأنه يختلف عن اتصال HTTPS.

إذا أجريت مقابلةً عمل من قبل، فقد تُسأَل ماذا يحدث عندما تكتب شيئًا ما في مربع بحث جوجل ثم تضغط مفتاح `Enter`؟ فهو أحد الأسئلة الأكثر شيوعًا التي ستُطرَح عليك، لمعرفة ما إذا كان بإمكانك شرح بعض المفاهيم الأساسية وما إذا كان لديك أيّ فكرة عن كيفية عمل الإنترنت، حيث سنحلّل ما يحدث عندما تكتب عنوان URL في شريط عنوان متصفحك ثم تضغط على `Enter`، وتُعَدّ هذه التقنية نادرة التغيّر وتشغّل أحد أكثر الأنظمة المجتمعية التي بناها الإنسان تعقيدًا واتساعًا.

6.2.1 تحليل طلبات URL

تملك المتصفحات الحديثة القدرة على معرفة ما إذا كان الشيء الذي كتبته في شريط العناوين هو عنوان URL فعلي أو مصطلح بحث، حيث سيستخدم المتصفح محرّك البحث الافتراضي إذا لم يكن عنوان URL صالحًا، فلنفترض أنك كتبتَ عنوان URL فعليًا، حيث ينشئ المتصفح أولًا عنوان URL الكامل عند إدخال العنوان ثم الضغط على مفتاح `Enter`، فإذا أدخلت نطاقًا مثل `flaviocopes.com`، فسيضيف المتصفح إلى بدايته `HTTP://` افتراضيًا اعتمادًا على بروتوكول HTTP.

> يجب عليك معرفة أنّ نظام ويندوز Windows قد يطبّق بعض الأشياء بطريقة مختلفة قليلًا عن نظامَي macOS ولينكس Linux.

6.2.2 مرحلة بحث DNS

يبدأ المتصفح عملية بحث DNS للحصول على عنوان IP الخادم، ويُعَدّ اسم النطاق اختصارًا مفيدًا للبشر، ولكن الإنترنت منظَّم بطريقة تمكّن الحواسيب من البحث عن موقع الخادم الدقيق من خلال عنوان IP الخاص به، وهو عبارة عن مجموعة من الأعداد مثل `222.324.3.1` في الإصدار IPv4، حيث يتحقق المتصفح أولًا من ذاكرة DNS المخبئية المحلية، للتأكد من أن النطاق قد جرى تحليله resolved مؤخرًا، كما يحتوي متصفح كروم Chrome على عارض مفيد لذاكرة DNS المخبئية الذي يمكنك رؤيته من خلال الرابط `chrome://net-internals/#dns`، فإذا لم تعثر على أي شيء هناك، فهذا يعني استخدام المتصفح محلّل DNS عن طريق استدعاء نظام POSIX `gethostbyname` لاسترداد معلومات المضيف.

gethostbyname

يبحث استدعاء النظام `gethostbyname` أولًا في ملف المضيفِين hosts المحلي، والذي يوجد في نظامَي macOS أو لينكس Linux ضمن `/etc/hosts`، للتأكد من أن النظام يوفّر المعلومات محليًا، فإذا لم

يقدّم ملف المضـيفِين المحلي أيّ معلومـات عن النطـاق، فسـيقدّم النظـام طلبًـا إلى خـادم DNS، حيث يُخـزَّن عنوان خادم DNS في تفضيلات النظام، كما يُعَدّ الخادمان التاليان خادمي DNS شهيرين:

- `8.8.8.8`: خادم DNS العام الخاص بجوجل.
- `1.1.1.1`: خادم CloudFlare DNS.

يستخدِم معظم الأشخاص خادم DNS الذي يوفِّره مزوّد خدمة الإنترنت الخـاص بهم، كمـا يطبّـق المتصـفح طلب DNS باسـتخدام بروتوكـول UDP، فـالبروتوكولان TCP وUDP من بروتوكـولات الشـبكات الحاسـوبية الأساسية ويتواجدان بالمستوى نفسه، لكن بروتوكول TCP موجَّه بالاتصال، بينما بروتوكول UDP عديم الاتصـال وأخف، ويُستخدَم لإرسال الرسائل مع قليل من الحِمل على الشبكة.

قد يحتوي خادم DNS على عنوان IP النطاق في الذاكرة المخبئية، فإن لم يكن كذلك، فسيسأل خـادم DNS الجذر، إذ يتكون هذا النظام من 13 خادم حقيقي موز ع في أنحاء العالم وهو يقود شبكة الإنترنت بأكملهـا، كمـا لا يعرف خادم DNS عنوان كل اسم نطـاق على هـذا الكـوكب، ولكن يكفيـه معرفـة مكـان وجـود محلّلي DNS من المسـتوى الأعلى، حيث يُعَـدّ نطـاق المسـتوى الأعلى top-level domain امتـداد النطـاق مثـل `.com` و `.it` و `.pizza` وغير ذلك.

يَعيد خادم DNS توجيه الطلب عند تلقّيه إلى خادم DNS الخاص بنطـاق المسـتوى الأعلى TLD، ولنفـترض أنك تبحث عن موقع `flaviocopes.com`، حيث يعيد خادم DNS الخاص بالنطـاق الجـذر عنـوان IP الخـاص بخادم نطاق المستوى الأعلى `.com`،، ويخزّن بعدها محلّـل DNS الخـاص بنـا عنـوان IP لخـادم نطـاق المسـتوى الأعلى، بحيث لا يتعيّن عليه أن يسأل خادم DNS الجذر مرةً أخرى عنه.

سيمتلك خادم DNS الخاص بنطاق المستوى الأعلى عناوين IP لخوادم الأسماء الرسـمية الخاصـة بالنطـاق الذي نبحث عنه، إذ عنـد شـرائك لنطـاقٍ يرسـل مسـجل النطـاق domain registrar نطـاق المسـتوى الأعلى المناسب TDL إلى خوادم الأسماء.، فإذا حدّثتَ خـوادم الأسـماء عنـد تغييـر مـزود الاستضـافة مثلًا، فسـيُحدِّث مسجّل النطاق الخاص بك هذه المعلومات تلقائيًا، ونوضِّح فيما يلي أمثلـةً عن خـوادم DNS لمـزود الاستضـافة التي تكون أكثر من خادم عادةً لاستخدامها على أساس نسخة احتياطية:

- `ns1.dreamhost.com`
- `ns2.dreamhost.com`
- `ns3.dreamhost.com`

يبدأ محلل DNS بالخادم الأول، ويحاول طلب عنوان IP الخـاص بالنطـاق مـع النطـاق الفـرعي أيضًـا الـذي تبحث عنه، وهو المصدر النهائي لعنوان IP.

6.2.3 إنشاء اتصال/مصافحة handshaking طلب TCP

يمكن للمتصفح الآن بدء اتصال TCP عند توفر عنوان IP الخادم، حيث يتطلب اتصال TCP عملية مصافحة handshaking قبل تهيئته بالكامل والبدء بإرسال البيانات، إذ يمكننا إرسال الطلب بعد إنشاء الاتصال.

6.2.4 إرسال الطلب

يكـون الطلب عبـارةً عن مسـتند نصـي منظَّم بطريقـة دقيقـة يحـدّدها بروتوكـول الاتصـال، ويتكـون من 3 أجزاء هي:

- سطر الطلب request line.
- ترويسة الطلب request header.
- جسم الطلب request body.

يضبط سطر الطلب ما يلي في سطر واحد:

- تابع HTTP.
- موقع المَورد.
- إصدار البروتوكول.

إليك المثال التالي:

```
GET / HTTP/1.1
```

تتكون ترويسة الطلب من مجموعة من أزواج الحقـل-القيمـة `field: value` الـتي تحـدِّد قيمًـا معينـةً، وهناك حقلان إلزاميان هما `Host` و`Connection`، بينما جميع الحقول الأخرى اختيارية:

```
Host: flaviocopes.com
Connection: close
```

يشير الحقل `Host` إلى اسم النطاق الذي نريد الوصول إليه، بينما يُضبَط الحقل `Connection` على القيمـة `close` دائمًا إلّا في حالة إبقاء الاتصال مفتوحًا، وبعض حقول الترويسة الأكثر استخدامًا هي:

- `Origin`
- `Accept`
- `Accept-Encoding`
- `Cookie`

- `Cache-Control`
- `Dnt`

وهناك غيرها الكثير، ويُنهَى جزء الترويسة بسطر فارغ.

أما جسم الطلب فهو اختياري ولا يُستخدَم في طلبات GET، ولكنه يُستخدَم بكثرة في طلبات POST وفي أفعال أخرى في بعض الأحيان، كمايمكن أن يحتوي على بيانات بتنسيق JSON، وبما أننا الآن نحلّل طلب GET، فإن الجسم فارغ.

6.2.5 الاستجابة Response

يعالِج الخادم الطلب بعد إرساله ويرسل استجابةً، حيث تبدأ الاستجابة برمز الحالة status code ورسالة الحالة status message، فإذا كان الطلب ناجحًا ويعيد القيمة 200، فستبدأ الاستجابة بما يلي:

```
200 OK
```

قد يعيد الطلب رمز ورسالة حالة مختلفَين مثل الأمثلة التالية:

```
404 Not Found
403 Forbidden
301 Moved Permanently
500 Internal Server Error
304 Not Modified
401 Unauthorized
```

تحتوي الاستجابة بعد ذلك على قائمة بترويسات HTTP وجسم الاستجابة الذي سيكون HTML لأننا ننفّذ الطلب في المتصفح.

6.2.6 تحليل HTML

تلقّى المتصفح الآن ملف HTML وبدأ في تحليله، وسيكرّر العملية نفسها بالضبط على جميع الموارد التي تطلبها الصفحة مثل:

- ملفات CSS.
- الصور.
- الأيقونة المفضلة أو رمز الموقع favicon.
- ملفات جافا سكريبت.
- وغير ذلك.

إنَّ الطريقة التي تصيّر render بها المتصفحاتُ الصفحةَ خارج نطاق مناقشتنا، ولكن يجب فهم أن العملية التي شرحناها هنا غير مقتصرة على صفحات HTML فقط، بل يمكن تطبيقها على أيّ عنصر مُقدَّم عبر بروتوكول HTTP.

6.2.7 بناء خادم HTTP باستخدام Node.js

خادم ويب HTTP الذي سنستخدِمه هو الخادم نفسه الذي استخدمناه سابقًا في تطبيق Hello World.

```
const http = require('http')
const port = 3000
const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})
server.listen(port, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

لنحلّل المثال السابق بإيجاز:

ضمّنا وحدة http التي نستخدمها لإنشاء خادم HTTP، وضُبِط الخادم للاستماع على المنفذ المحدَّد 3000، حيث تُستدعَى دالة رد النداء listen عندما يكون الخادم جاهزًا، فدالة رد النداء التي نمررها هي الدالة التي ستُنفَّذ عند وصول كل طلب، ويُستدعَى حدث request عند تلقّي طلب جديد، مما يوفّر كائنين هما: طلب (كائن http.IncomingMessage) واستجابة (كائن http.ServerResponse).

يوفّر الطلب request تفاصيل الطلب، حيث نصل من خلاله إلى ترويسات الطلبات وبياناتها؛ أما الاستجابة response فتُستخدَم لتوفير البيانات التي سنعيدها إلى العميل، كما ضبطنا خاصية statusCode على القيمة 200 في مثالنا، للإشارة إلى استجابة ناجحة.

```
res.statusCode = 200
```

وضبطنا ترويسة Content-Type كما يلي:

```
res.setHeader('Content-Type', 'text/plain')
```

ثم أغلقنا الاستجابة في النهاية بإضافة المحتوى على أساس وسيط للتابع end():

```
res.end('Hello World\n')
```

6.2.8 إجراء طلبات HTTP

سنشرح كيفية إجراء طلبات HTTP في Node.js باستخدام GET و POST و PUT و DELETE.

> سنستخدم مصطلح HTTP، ولكن HTTPS هو ما يجب استخدامه في كل مكان، لذلك تستخدِم هذه الأمثلة بروتوكول HTTPS بدلًا من HTTP.

ا. إجراء طلب GET

```
const https = require('https')
const options = {
  hostname: 'flaviocopes.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}
const req = https.request(options, (res) => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    process.stdout.write(d)
  })
})
req.on('error', (error) => {
  console.error(error)
})
req.end()
```

ب. إجراء طلب POST

```
const https = require('https')
const data = JSON.stringify({
  todo: 'Buy the milk'
})
const options = {
  hostname: 'flaviocopes.com',
  port: 443,
  path: '/todos',
```

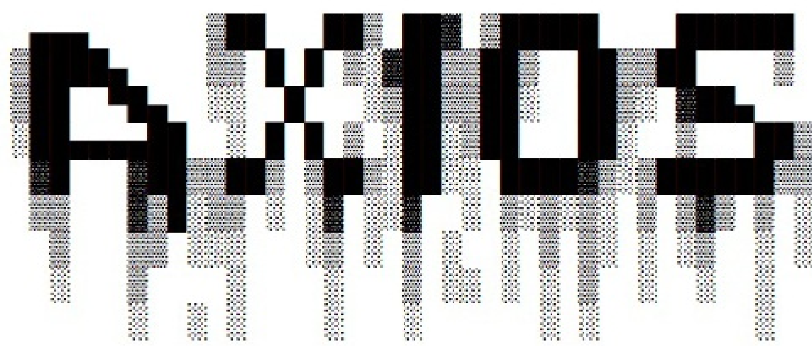
```
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Content-Length': data.length
    }
  }
  const req = https.request(options, (res) => {
    console.log(`statusCode: ${res.statusCode}`)

    res.on('data', (d) => {
      process.stdout.write(d)
    })
  })
  req.on('error', (error) => {
    console.error(error)
  })
  req.write(data)
  req.end()
```

ج. PUT و DELETE

تستخدِم طلبات PUT و DELETE تنسيق طلب POST نفسه مع تغيير قيمة `options.method` فقط.

6.3 مكتبة Axios

تُعَدّ Axios مكتبة جافا سكريبت يمكنك استخدامها من أجل إجراء طلبات HTTP، وهي تعمل في المنصَّتين: المتصفح Browser وبيئة نود جي إس Node.js.



تدعم هذه المكتبة جميع المتصفحات الحديثة بما في ذلك الإصدار IE8 والإصدارات الأحدث، كما تستند على الوعود، وهذا يتيح لنا كتابة شيفرة صيغة عدم التزامن أو الانتظار async/await لإجراء طلبات XHR

بسهولة، كما يتمتع استخدام مكتبة Axios ببعض المزايا بالمقارنة مع واجهة API Fetch الأصيلة، وهذه المزايـا هي كالتالي:

- تدعم المتصفحات القديمة، حيث تحتاج Fetch إلى تعويض نقص دعم المتصفحات polyfill.
- لديها طريقة لإبطال طلب.
- لديها طريقة لضبط مهلة الاستجابة الزمنية.
- تحتوي على حماية CSRF مبنية مسبقًا.
- تدعم تقدّم التحميل.
- تجري تحويل بيانات JSON تلقائيًا.
- تعمل في Node.js.

6.3.1 تثبيت Axios

يمكن تثبيت Axios باستخدام npm:

```
npm install axios
```

أو باستخدام yarn:

```
yarn add axios
```

أو يمكنك تضمينها ببساطة في صفحتك باستخدام unpkg.com كما يلي:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

6.3.2 واجهة برمجة تطبيقات Axios

يمكنك بدء طلب HTTP من كائن axios:

```
axios({
  url: 'https://dog.ceo/api/breeds/list/all',
  method: 'get',
  data: {
    foo: 'bar'
  }
})
```

لكنك ستستخدم التوابع التالية كما هو الحال في jQuery، حيث يمكنك استخدام `$.get()` و `$.post()` بدلًا من `$.ajax()`:

- `axios.get()`
- `axios.post()`

توفِّر مكتبة Axios توابعًا لجميع أفعال HTTP، والتي تُعَدّ أقل شيوعًا ولكنها لا تزال مُستخدَمة:

- `axios.delete()`
- `axios.put()`
- `axios.patch()`
- `axios.options()`
- `axios.head()`: وهو تابع يُستخدَم للحصول على ترويسات HTTP لطلب ما مع تجاهل الجسم.

6.3.3 إرسال واستقبال الطلبات

إحدى الطرق الملائمة لاستخدام مكتبة Axios هي استخدام صيغة async/await الحديثة في الإصدار ES2017، حيث يستعمل مثال Node.js التالي عن واجهة Dog API لاسترداد قائمة بجميع سلالات الكلاب dogs breeds باستخدام التابع `axios.get()`، ويحصي هذه السلالات:

```
const axios = require('axios')
const getBreeds = async () => {
  try {
    return await axios.get('https://dog.ceo/api/breeds/list/all')
  } catch (error) {
    console.error(error)
  }
}
const countBreeds = async () => {
  const breeds = await getBreeds()

  if (breeds.data.message) {
    console.log(`Got ${Object.entries(breeds.data.message).length}
breeds`)
  }
}
countBreeds()
```

إذا لم ترغب في استخدام صيغة async/await، فيمكنك استخدام صيغة الوعود Promises:

```
const axios = require('axios')
const getBreeds = () => {
  try {
    return axios.get('https://dog.ceo/api/breeds/list/all')
  } catch (error) {
    console.error(error)
  }
}
const countBreeds = async () => {
  const breeds = getBreeds()
    .then(response => {
      if (response.data.message) {
        console.log(
          `Got ${Object.entries(response.data.message).length} breeds`
        )
      }
    })
    .catch(error => {
      console.log(error)
    })
}
countBreeds()
```

يمكن أن تحتوي استجابة GET على معامِلات في عنـوان URL مثـل `https://site.com/?foo=bar`، حيث يمكنك تطبيق ذلك في مكتبة Axios عن طريق استخدام عنوان URL كما يلي:

```
axios.get('https://site.com/?foo=bar')
```

أو يمكنك استخدام خاصية `params` في الخيارات كما يلي:

```
axios.get('https://site.com/', {
  params: {
    foo: 'bar'
  }
})
```

يشبه إجراء طلب POST تمامًا إجراء طلب GET مع استخدم `axios.post` بدلًا من `axios.get`:

```
axios.post('https://site.com/')
```

الكائن الذي يحتوي على معامِلات POST هو الوسيط الثاني:

```
axios.post('https://site.com/', {
  foo: 'bar'
})
```

6.4 مقابس الويب Websockets

مقابس الويب WebSockets هي بديل لاتصال HTTP في تطبيقات الـويب، إذ تـوفِّر قنـاة اتصـال ثنائيـة الاتجاه طويلة الأمد بين العميل والخادم، كما تبقى القناة مفتوحة بمجرد إنشائها، مما يوفر اتصالًا سريعًا جدًا مـع زمن انتقال وحِمل منخفضَين، كما تدعم جميع المتصفحات الحديثة مقابس WebSockets.

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *
		53	59	8	44	8.4
6		54	60	9	45	9.2
7	12	55	61	9.1	46	9.3
8	13	56	62	10	47	10.2
9	14	57	63	10.1	48	10.3
10	15	58	64	11	49	11.2
11	16	59	65	11.1	50	11.3
	17	60	66	TP	51	
	18	61	67		52	
			68			

قد تتساءل، ما وجه الاختلاف بين WebSockets وبين HTTP؟ حسنًا، يُعَدّ HTTP بروتوكولًا وطريقة تواصل مختلفة تمامًا، فهو بروتوكول طلب/استجابة أو request/response، حيث يعيد الخادم البيانات الـتي يطلبهـا العميل، بينما تفيد مقابس WebSockets في الحالات التالي:

- يمكن للخادم إرسال رسالة إلى العميل دون أن يطلب العميل صراحةً شيئًا ما.
- يمكن للعميل والخادم التحدث مع بعضهما البعض في الوقت نفسه.
- في حالـة تبـادل كميـة قليلـة من البيانـات الإضـافية لإرسـال الرسـائل، وهـذا يعـني اتصـالًا ذا زمن انتقال منخفض.

تُعَدّ مقابس WebSockets مناسبةً للاتصالات طويلة الأمد في الوقت الحقيقي، بينما يُعَدّ بروتوكول HTTP مفيدًا لتبادل البيانات والتفاعلات المؤقتة التي يبدأها العميل، كما يُعَدّ بروتوكول HTTP أبسط بكثير في التطبيق، بينما تتطلب مقابس WebSockets مزيدًا من العبء الإضافي.

6.4.1 مقابس الويب الآمنة

استخدم دائمًا البروتوكول الآمن والمشفّر لمقابس الويب أي `wss://`، ويشير `ws://` إلى إصدار مقابس WebSockets غير الآمن -مثل `http://` في مقابس WebSockets- الذي يجب تجنبه.

6.4.2 إنشاء اتصال WebSockets جديد

إليك المثال التالي:

```
const url = 'wss://myserver.com/something'
const connection = new WebSocket(url)
```

يُعَدّ `connection` كائن `WebSocket`، كما يُشغَّل حدث `open` عند إنشاء الاتصال بنجاح، ويمكنك الاستماع إلى الاتصال عن طريق إسناد دالة رد نداء callback إلى خاصية `onopen` الخاصة بكائن `connection` كما يلي:

```
connection.onopen = () => {
  //...
}
```

إذا كان هناك أي خطأ، فستُشغَّل دالة رد النداء `onerror` كما يلي:

```
connection.onerror = error => {
  console.log(`WebSocket error: ${error}`)
}
```

6.4.3 إرسال البيانات إلى الخادم باستخدام WebSockets

يمكنك إرسال البيانات إلى الخادم بمجرد فتح الاتصال، حيث يمكنك إرسال البيانات بسهولة ضمن دالة رد النداء `onopen` كما يلي:

```
connection.onopen = () => {
  connection.send('hey')
}
```

6.4.4 استقبال البيانات من الخادم باستخدام WebSockets

استمع إلى الاتصال باستخدام دالة رد النداء onmessage التي تُستدعَى عند تلقي حدث message كما يلي:

```
connection.onmessage = e => {
  console.log(e.data)
}
```

6.4.5 تطبيق خادم WebSockets في Node.js

تُعَدّ مكتبة ws مكتبة WebSockets شائعةً ومُستخدَمةً مع Node.js، كما سنستخدمها لبناء خادم WebSockets، ويمكن استخدامها أيضًا لتطبيق العميل مع استخدام مقابس WebSockets للتواصل بين خدمَتين من الخدمات الخلفية، كما يمكنك تثبيت هذه المكتبة بسهولة باستخدام الأمر التالي:

```
yarn init
yarn add ws
```

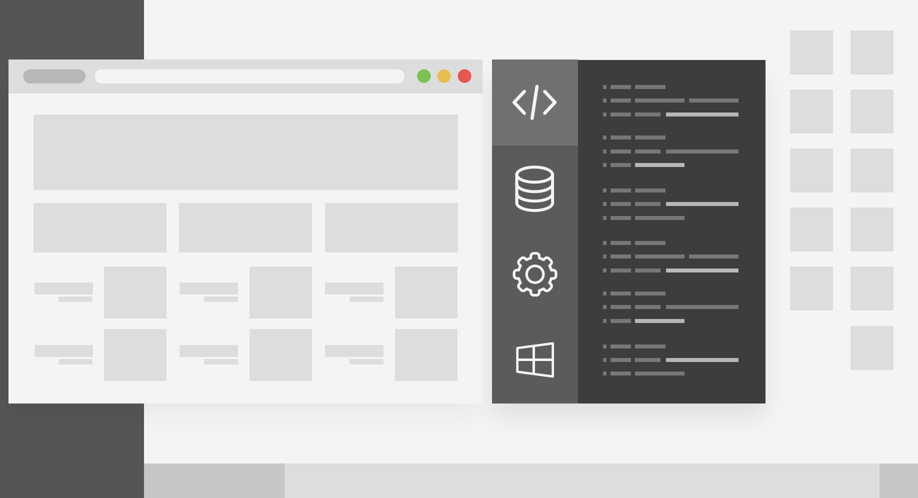
الشيفرة التي تحتاج إلى كتابتها ليست كبيرةً وهي كما يلي:

```
const WebSocket = require('ws')
const wss = new WebSocket.Server({ port: 8080 })
wss.on('connection', ws => {
  ws.on('message', message => {
    console.log(`Received message => ${message}`)
  })
  ws.send('ho!')
})
```

تُنشئ الشيفرة السابقة خادمًا جديدًا على المنفذ 8080 وهو المنفذ الافتراضي لمقابس الويب WebSockets، وتضيف دالة رد نداء عند إنشاء اتصال، مما يؤدي إلى إرسال !ho إلى العميل، وتسجيل الرسائل التي يتلقاها.

شاهد مثالًا حيًا لخادم مقابس الويب WebSockets ومثالًا حيًا لعميل WebSockets يتفاعل مع الخادم على Glitch.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

7. التعامل مع الملفات في Node.js

سنتعرّف من خلال هذا الفصل على كيفية التعامل مـع الملفـات والمجلـدات في Node.js من خلال شـرح واصفات الملفـات وإحصـائياتها ومسـاراتها وقراءتهـا وكتابتهـا، كمـا سـنتعرّف على مفهـوم المجـاري streams وميّزاتها وأنواعها.

7.1 واصفات الملفات File descriptors

يمكن التفاعل مع واصـفات الملفـات باسـتخدام نـود Node، إذ يجب أن تحصـل على واصـف ملـف قبـل تمكنك من التفاعل مع ملف موجود في نظام الملفات الخاص بك، وواصف الملف هو ما يُعاد عند فتح الملــف باستخدام التابع open() الذي توفره وحدة fs:

```
const fs = require('fs')
fs.open('/Users/flavio/test.txt', 'r', (err, fd) => {
  //الملف واصف هو fd
})
```

استخدمنا الراية r على أساس معامِل ثـانٍ لاسـتدعاء fs.open()، إذ تعـني هـذه الرايـة أننـا نفتح الملـف للقراءة؛ أما الرايات الأخرى المُستخدَمة فهي:

- r+ فتح الملف للقراءة والكتابة.
- w+ فتح الملف للقراءة والكتابة، مع وضع المجرى stream في بداية الملـف وإنشـاء الملـف إذا لم يكن موجودًا مسبقًا.
- a فتح الملف للكتابة، مع وضع المجرى في نهاية الملف وإنشاء الملف إن لم يكن موجودًا مسبقًا.

- `a+` فتح الملف للقراءة والكتابة، مع وضع المجرى في نهاية الملف وإنشاء الملف إن لم يكن موجودًا مسبقًا.

يمكن فتح الملف باستخدام التابع `fs.openSync` الذي يعيد كائن واصف الملف بدل توفيره بدالة رد نداء:

```
const fs = require('fs')
try {
  const fd = fs.openSync('/Users/flavio/test.txt', 'r')
} catch (err) {
  console.error(err)
}
```

يمكنك تنفيذ جميع العمليات المطلوبة مثل استدعاء التابع `fs.open()` والعديد من العمليات الأخرى التي تتفاعل مع نظام الملفات بمجرد حصولك على واصف الملف بأيّ طريقة تختارها.

7.2 إحصائيات الملف

يأتي كل ملف مع مجموعة من التفاصيل التي يمكننا فحصها باستخدام نود Node باستخدام التابع `stat()` الذي توفِّره وحدة `fs`، حيث يمكنك استدعاؤه مع تمرير مسار ملف إليه، حيث سيستدعي نود بعد حصوله على تفاصيل الملف دالة رد النداء التي تمررها مع معاملين هما رسالة خطأ وإحصائيات الملف:

```
const fs = require('fs')
fs.stat('/Users/flavio/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }
  //`stats` يمكننا الوصول إلى إحصائيات الملف في
})
```

كما يوفِّر نود تابعًا متزامنًا يوقِف الخيط thread إلى أن تصبح إحصائيات الملف جاهزةً:

```
const fs = require('fs')
try {
  const stats = fs.stat('/Users/flavio/test.txt')
} catch (err) {
  console.error(err)
}
```

تُضمَّن معلومات الملف في المتغير `stats`، ويمكننا استخراج أنواع معلومات متعددة باستخدام توابع stats كما يلي:

- استخدم التابع `stats.isFile()` والتابع `stats.isDirectory()` لمعرفة إذا كان الملف عبارة عن مجلد أو ملف.
- استخدم التابع `stats.isSymbolicLink()` لمعرفة إذا كان الملف وصلةً رمزيةً symbolic link.
- استخدم التابع `stats.size` لمعرفة حجم الملف مقدَّرًا بالبايت.

هناك توابع متقدمة أخرى، ولكن الجزء الأكبر مما ستستخدمه هو التوابع السابقة.

```
const fs = require('fs')
fs.stat('/Users/flavio/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }
  stats.isFile() //true
  stats.isDirectory() //false
  stats.isSymbolicLink() //false
  stats.size //1024000 //= 1MB
})
```

7.3 مسارات الملفات

سنتعرّف على كيفية التفاعل مع مسارات الملفات والتعامل معها في نود Node، فلكل ملف في النظام مسار، وقد يبدو المسار في نظامَي لينكس Linux وmacOS كما يلي:

```
/users/flavio/file.txt
```

بينما الحواسيب التي تعمل بنظام ويندوز Windows مختلفة، إذ يكون للمسار بنية كما يلي:

```
C:\users\flavio\file.txt
```

يجب الانتباه عند استخدام المسارات في تطبيقاتك، إذ يجب مراعاة هذا الاختلاف، كما يمكنك تضمين وحدة المسار `path` في ملفاتك كما ما يلي:

```
const path = require('path')
```

ثم يمكنك البدء في استخدام توابعها، كما يمكنك استخراج معلومات من مسار باستخدام التوابع التالية:

- `dirname`: للحصول على مجلد الملف الأب.
- `basename`: للحصول على جزء اسم الملف.
- `extname`: للحصول على امتداد الملف.

إليك المثال التالي:

```
const notes = '/users/flavio/notes.txt'
path.dirname(notes) // /users/flavio
path.basename(notes) // notes.txt
path.extname(notes) // .txt
```

يمكنك الحصول على اسم الملف بدون امتداده عن طريق تحديد وسيط ثانٍ للتابع `basename` كما يلي:

```
path.basename(notes, path.extname(notes)) //notes
```

كما يمكنك ربط جزأين أو أكثر من المسار مع بعضها البعض باستخدام التابع `path.join()` كما يلي:

```
const name = 'flavio'
path.join('/', 'users', name, 'notes.txt') //'/users/flavio/notes.txt'
```

يمكنك حساب مسار الملف المطلق absolute path من مساره النسبي relative path باستخدام التابع `path.resolve()` كما يلي:

```
path.resolve('flavio.txt') //'/Users/flavio/flavio.txt' if run from my home folder
```

سيُلحِق في هذه الحالة نود Node ببساطة المسارَ النسبي `/flavio.txt` بدليل أو مجلد العمل الحالي، فإذا حددت مجلدًا على أساس معامل آخر، فسيستخدِم تابع `resolve` المعامل الأول أساسًا للمعامل الثاني كما يلي:

```
path.resolve('tmp', 'flavio.txt')// إذا '/Users/flavio/tmp/flavio.txt' شُغِّل من المجلد المحلي
```

إذا بدأ المعامل الأول بشرطة مائلة، فهذا يعني أنه مسار مطلق مثل المثال التالي:

```
path.resolve('/etc', 'flavio.txt')//'/etc/flavio.txt'
```

يُعَدّ `path.normalize()` تابعًا آخرًا مفيدًا يحسب المسار الفعلي عندما يحتوي على محددات نسبية مثل `.` أو `..` أو شرطة مائلة مزدوجة كما يلي:

```
path.normalize('/users/flavio/..//test.txt') ///users/test.txt
```

لن يتحقق التابعان `resolve` و`normalize` من وجود المسار، وإنما يحسبان المسار فقط بناءً على المعلومات المتاحة.

7.4 قراءة الملفات

أبسط طريقة لقراءة ملف في نود هي استخدام تابع `fs.readFile()`، حيث نمرِّر له مسار الملف ودالة رد النداء التي ستُستدعَى مع بيانات الملف ومع الخطأ كما يلي:

```
const fs = require('fs')
fs.readFile('/Users/flavio/test.txt', (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

يمكنك بدلًا من ذلك استخدام الإصدار المتزامن من التابع السابق وهو التابع `fs.readFileSync()`:

```
const fs = require('fs')

try {
  const data = fs.readFileSync('/Users/flavio/test.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

الترميز الافتراضي هو utf8، ولكن يمكنك تحديد ترميز مُخصَّص باستخدام معامل ثانٍ، كما يقرأ كل من التابعَين `fs.readFile()` و`fs.readFileSync()` محتوى الملف الكامل في الذاكرة قبل إعادة البيانات، وهذا يعني أن الملفات الكبيرة سيكون لها تأثير كبير على استهلاك الذاكرة وسرعة تنفيذ البرنامج، وبالتالي يكون الخيار الأفضل في هذه الحالة هو قراءة محتوى الملف باستخدام المجاري streams.

7.5 كتابة الملفات

أسهل طريقة للكتابة في الملفات في Node.js هي استخدام واجهة برمجة تطبيقات `fs.writeFile()`، وإليك المثال التالي:

```
const fs = require('fs')
const content = 'Some content!'
fs.writeFile('/Users/flavio/test.txt', content, (err) => {
  if (err) {
    console.error(err)
    return
  }
  //كُتِب الملف بنجاح
})
```

يمكنك بدلًا من ذلك استخدام الإصدار المتزامن وهو `fs.writeFileSync()`:

```
const fs = require('fs')
const content = 'Some content!'
try {
  const data = fs.writeFileSync('/Users/flavio/test.txt', content)
  //كُتِب الملف بنجاح
} catch (err) {
  console.error(err)
}
```

ستبدّل واجهة برمجة التطبيقات هذه افتراضيًا محتويات الملف إذا كان موجودًا مسبقًا، ولكن يمكنك تعديل الإعداد الافتراضي عن طريق تحديد راية كما يلي:

```
fs.writeFile('/Users/flavio/test.txt', content, { flag: 'a+' }, (err)
=> {})
```

الرايات التي يمكنك استخدامها هي:

- `r+` لفتح الملف للقراءة والكتابة.
- `w+` لفتح الملف للقراءة والكتابة، مع وضع المجرى ببداية الملف وإنشاء الملف إن لم يكن موجودًا.
- `a` لفتح الملف للكتابة، مع وضع المجرى في نهاية الملف وإنشاء الملف إذا لم يكن موجودًا مسبقًا.
- `a+` لفتح الملف للقراءة والكتابة، مع وضع المجرى في نهاية الملف، وإنشاء الملف إن لم يكن موجودًا.

يمكنك العثور على المزيد من الرايات على nodejs.

7.6 إلحاق محتوى بملف

يمكنك إلحاق محتوى بنهاية الملف من خلال استخدام التابع `fs.appendFile()` ونسخته المتزامنة التابع `fs.appendFileSync()`:

```
const content = 'Some content!'
fs.appendFile('file.log', content, (err) => {
  if (err) {
    console.error(err)
    return
  }
  //done!
})
```

7.7 استخدام المجاري streams

تكتب كل التوابع السابقة المحتوى الكامل في الملف قبل إعادة التحكم إلى برنامجك مرةً أخرى، أي تنفيذ دالة رد النداء في النسخة غير المتزامنة، وبالتالي الخيار الأفضل هو كتابة محتوى الملف باستخدام المجاري streams.

لنتعرّف على الغرض الأساسي من المجاري streams وسبب أهميتها وكيفية استخدامها، حيث سنقدِّم مدخلًا بسيطًا إلى المجاري، ولكن هناك جوانب أكثر تعقيدًا لتحليلها.

7.7.1 مفهوم المجاري streams

تُعَدّ المجاري أحد المفاهيم الأساسية التي تعمل على تشغيل تطبيقات Node.js، وهي طريقة للتعامل مع ملفات القراءة/الكتابة أو اتصالات الشبكة أو أيّ نوع من تبادل المعلومات من طرف إلى طرف بطريقة فعالة، وليست المجاري مفهومًا خاصًا بنود Node.js، فقد توفّرت في نظام التشغيل يونيكس Unix منذ عقود، ويمكن للبرامج أن تتفاعل مع بعضها البعض عبر تمرير المجاري من خلال معامِل الشريط العمودي أو الأنبوب pipe operator (`|`).

يُقرأ الملف في الذاكرة من البداية إلى النهاية ثم يعالج، عندما تطلب من البرنامج قراءة ملف بالطريقة التقليدية، لكن يمكنك قراءة الملف قطعةً تلو الأخرى باستخدام المجاري، ومعالجة محتواه دون الاحتفاظ به بالكامل في الذاكرة، إذ توفِّر وحدة نود stream الأساس الذي يُبنَى عليه جميع واجهات برمجة التطبيقات ذات المجرى، كما توفِّر المجاري ميزتَين رئيسيتَين باستخدام طرق معالجة البيانات الأخرى هما:

- **فعالية الذاكرة Memory efficiency**: لست بحاجة إلى تحميل كميات كبيرة من البيانات في الذاكرة قبل أن تكون قادرًا على معالجتها.
- **فعالية الوقت Time efficiency**: تستغرق وقتًا أقل لبدء معالجة البيانات بمجرد حصولك عليها، بدلًا من انتظار اكتمال حمولة البيانات للبدء.

يوضِّح المثال التالي قراءة ملفات من القرص الصلب، حيث يمكنك باستخدام وحدة نود fs قراءة ملف وتقديمه عبر بروتوكول HTTP عند إنشاء اتصال جديد بخادم http:

```
const http = require('http')
const fs = require('fs')
const server = http.createServer(function (req, res) {
  fs.readFile(__dirname + '/data.txt', (err, data) => {
    res.end(data)
  })
})
server.listen(3000)
```

يقرأ التابع `readFile()` محتويات الملف الكاملة، ويستدعي دالة رد النداء callback function عند الانتهاء، بينما سيعيد التابع `res.end(data)` في دالة رد النداء محتويات الملف إلى عميل HTTP، فإذا كان الملف كبيرًا، فستستغرق العملية وقتًا طويلًا، ويمكن تطبيق الأمر نفسه باستخدام المجاري streams كما يلي:

```
const http = require('http')
const fs = require('fs')
const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(__dirname + '/data.txt')
  stream.pipe(res)
})
server.listen(3000)
```

يمكننا بث الملف عبر المجاري إلى عميل HTTP بمجرد أن يكون لدينا مجموعة كبيرة من البيانات جاهزة للإرسال بدلًا من انتظار قراءة الملف بالكامل؛ ويستخدم المثال السابق `stream.pipe(res)`، أي استدعاء تابع `pipe()` في مجرى الملف، حيث يأخذ هذا التابع المصدر، ويضخّه إلى وجهة معينة، كما يُستدعَى هذا التابع على مجرى المصدر، وبالتالي يُضَخ مجرى الملف إلى استجابة HTTP في هذه الحالة، وتكون القيمة المُعادة من التابع `pipe()` هي مجرى الوجهة، وهذا أمر ملائم للغاية لربط استدعاءات `pipe()` متعددة كما يلي:

```
src.pipe(dest1).pipe(dest2)
```

الذي يكافئ ما يلي:

```
src.pipe(dest1)
dest1.pipe(dest2)
```

تتوفَّر واجهات برمجة تطبيقات API الخاصة بنود Node التي تعمل باستخدام المجاري Streams، إذ تــوفِّر العديد من وحدات Node.js الأساسية إمكانات معالجة المجرى الأصيلة، ومن أبرزها:

- `process.stdin` التي تعيد مجرًى متصلًا بمجرى stdin.
- `process.stdout` التي تعيد مجرًى متصلًا بمجرى stdout.
- `process.stderr` التي تعيد مجرًى متصلًا بمجرى stderr.
- `fs.createReadStream()` الذي ينشئ مجرًى قابلًا للقراءة إلى ملف.
- `fs.createWriteStream()` الذي ينشئ مجرًى قابلًا للكتابة إلى ملف.
- `net.connect()` الذي يبدأ اتصالًا قائمًا على مجرى.
- `http.request()` الــذي يعيــد نســخة من الصــنف `http.ClientRequest`، وهــو مجــرى قابل للكتابة.
- `zlib.createGzip()` الذي يضغط البيانات باستخدام خوارزمية الضغط gzip في مجرى.
- `zlib.createGunzip()` الذي يفك ضغط مجرى gzip.
- `zlib.createDeflate()` الذي يضغط البيانات باستخدام خوارزمية الضغط deflate في مجرى.
- `zlib.createInflate()` الذي يفك ضغط مجرى deflate.

7.7.2 أنواع المجاري المختلفة

هناك أربع أصناف من المجاري هي:

- `Readable`: هو مجرى يمكن الضخ pipe منه ولكن لا يمكن الضخ إليه، أي يمكنك تلقي البيانـات منـه ولكن لا يمكنك إرسال البيانات إليه، فإذا دفعتَ بيانات إلى مجرى قابل للقراءة، فسـتُخزَّن مؤقتًـا حـتى يبدأ المستهلك في قراءة البيانات.
- `Writable`: هو مجرى يمكن الضخ إليه، ولكن لا يمكن الضـخ منـه، أي يمكنـك إرسـال البيانـات إليـه، ولكن لا يمكنك تلقي البيانات منه.

- Duplex: هو مجرى يمكن الضخ منه وإليه، أي هو مزيج من مجرى Readable ومجرى Writable.
- Transform: مجرى التحويل مشابه للمجرى Duplex، ولكن خرجه هو تحويل لدخله.

7.7.3 كيفية إنشاء مجرى قابل للقراءة

يمكن الحصول على مجرى قابل للقراءة من وحدة stream، كما يمكن تهيئته كما يلي:

```
const Stream = require('stream')
const readableStream = new Stream.Readable()
```

ثم يمكننا إرسال البيانات إليه بعد تهيئته:

```
readableStream.push('hi!')
readableStream.push('ho!')
```

7.7.4 كيفية إنشاء مجرى قابل للكتابة

يمكنك إنشاء مجرى قابل للكتابة من خلال وراثة كائن Writable الأساسي وتطبيق تابعه _write().

أنشئ أولًا كائن Stream كما يلي:

```
const Stream = require('stream')
const writableStream = new Stream.Writable()
```

ثم التابع _write كما يلي:

```
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}
```

يمكنك الآن الضخ إلى مجرى قابل للقراءة كما يلي:

```
process.stdin.pipe(writableStream)
```

7.7.5 كيفية الحصول على بيانات من مجرى قابل للقراءة

يمكنك قراءة البيانات من مجرًى قابل للقراءة باستخدام مجرى قابل للكتابة كما يلي:

```
const Stream = require('stream')
const readableStream = new Stream.Readable()
```

```
const writableStream = new Stream.Writable()
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}
readableStream.pipe(writableStream)
readableStream.push('hi!')
readableStream.push('ho!')
```

كما يمكنك استهلاك مجرى قابل للقراءة مباشرةً باستخدام الحدث `readable` كما يلي:

```
readableStream.on('readable', () => {
  console.log(readableStream.read())
})
```

7.7.6 كيفية إرسال بيانات إلى مجرى قابل للكتابة

استخدم تابع المجرى `write()` كما يلي:

```
writableStream.write('hey!\n')
```

7.7.7 إعلام مجرى قابل للكتابة بانتهاء الكتابة

استخدم التابع `end()` كما يلي:

```
const Stream = require('stream')
const readableStream = new Stream.Readable()
const writableStream = new Stream.Writable()
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}
readableStream.pipe(writableStream)
readableStream.push('hi!')
readableStream.push('ho!')
writableStream.end()
```

7.8 التعامل مع المجلدات

توفِّر وحدة Node.js الأساسية `fs` توابعًا متعددةً مفيدةً يمكنك استخدامها للتعامل مع المجلدات.

7.8.1 التحقق من وجود مجلد

يُستخدَم التابع `fs.access()` للتحقق مما إذا كان المجلد موجودًا، ويمكن لنود الوصول إلى المجلد باستخدام أذوناته.

7.8.2 إنشاء مجلد جديد

يُستخدَم التابع `fs.mkdir()` أو التابع `fs.mkdirSync()` لإنشاء مجلد جديد.

```
const fs = require('fs')
const folderName = '/Users/flavio/test'
try {
  if (!fs.existsSync(dir)){
    fs.mkdirSync(dir)
  }
} catch (err) {
  console.error(err)
}
```

7.8.3 قراءة محتوى مجلد

يُستخدَم التابع `fs.readdir()` أو التابع `fs.readdirSync` لقراءة محتويات مجلد، ويقرأ جزء الشيفرة التالية محتوى مجلد من ملفات ومجلدات فرعية، ويعيد مساراتها النسبية:

```
const fs = require('fs')
const path = require('path')
const folderPath = '/Users/flavio'
fs.readdirSync(folderPath)
```

يمكنك الحصول على المسار الكامل من خلال ما يلي:

```
fs.readdirSync(folderPath).map(fileName => {
  return path.join(folderPath, fileName)
}
```

كما يمكنك تصفية النتائج لإعادة الملفات فقط واستبعاد المجلدات كما يلي:

```
const isFile = fileName => {
  return fs.lstatSync(fileName).isFile()
}
fs.readdirSync(folderPath).map(fileName => {
  return path.join(folderPath, fileName)).filter(isFile)
}
```

7.8.4 إعادة تسمية مجلد

يُستخدَم التابع `fs.rename()` أو التابع `fs.renameSync()` لإعادة تسمية مجلد، حيث يكون المعامِل الأول هو المسار الحالي، والمعامِل الثاني هو المسار الجديد:

```
const fs = require('fs')
fs.rename('/Users/flavio', '/Users/roger', (err) => {
  if (err) {
    console.error(err)
    return
  }
  //done
})
```

كما يمكنك استخدام التابع `fs.renameSync()` الذي هو النسخة المتزامنة كما يلي:

```
const fs = require('fs')
try {
  fs.renameSync('/Users/flavio', '/Users/roger')
} catch (err) {
  console.error(err)
}
```

7.8.5 إزالة مجلد

يُستخدَم التابع `fs.rmdir()` أو التابع `fs.rmdirSync()` لإزالة مجلد، ويمكن أن تكون إزالة مجلد أكثر تعقيدًا إذا تضمّن محتوىً، لذلك نوصي في هذه الحالة بتثبيت وحدة fs-extra التي تحظى بشعبية ودعم كبيرَين، وهي بديل سريع لوحدة `fs`، وبالتالي تضيف مزيدًا من الميزات عليها، كما ستحتاج استخدام التابع `remove()`.

ثبّت وحدة `fs-extra` باستخدام الأمر: `npm install fs-extra`، واستخدمها كما يلي:

```
const fs = require('fs-extra')
const folder = '/Users/flavio'
fs.remove(folder, err => {
  console.error(err)
})
```

كما يمكن استخدامها مع الوعود promises كما يلي:

```
fs.remove(folder).then(() => {
  //done
}).catch(err => {
  console.error(err)
})
```

أو مع صيغة async/await كما يلي:

```
async function removeFolder(folder) {
  try {
    await fs.remove(folder)
    //done
  } catch (err) {
    console.error(err)
  }
}
  const folder = '/Users/flavio'
  removeFolder(folder)
```

للمزيد، يمكنك الرجوع إلى توثيق التعامل مع نظام الملفات في Node.js في موسوعة حسوب.

دورة تطوير واجهات المستخدم

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

8. تعرف على وحدات Node.js الأساسية

سنتعرّف من خلال هذا الفصل على وحدات Node.js الأساسية مثل وحدة fs و path و os و events وتوابعها المتعددة، كما يمكنك بناء وحدة مخصَّصة بالاعتماد على الوحدات الأساسية، حيث سنتعرّف على كيفية استخدام واجهة module.exports البرمجية لتصدير بياناتك، وسنتعرّف على وحدة MySQL للتعامل مع قواعد البيانات.

8.1 وحدة fs

توفّر وحدة `fs` عمليات متعددة ومفيدة للوصول إلى نظام الملفات والتفاعل معه، وليست هناك حاجة لتثبيتها نظرًا لكونها جزءًا من نواة نود، إذ يمكن استخدامها ببساطة عن طريق طلبها كما يلي:

```
const fs = require('fs')
```

ثم يمكنك الوصول إلى جميع توابعها التي تشمل ما يلي:

- `fs.access()`: يتحقق من وجود الملف ويمكن لنود الوصول إلى الملف باستخدام أذوناته.
- `fs.appendFile()`: يُلحِق بيانات بملف، وينشئ الملف إذا كان غير موجود مسبقًا.
- `fs.chmod()`: يغيّر أذونات الملف المحدَّد بواسطة اسم الملف المُمرَّر، ويتعلق بالتابعَين `fs.lchmod()` و `fs.fchmod()`.
- `fs.chown()`: يغيّر مالك ومجموعة الملف المحدَّد بواسطة اسم الملف المُمرَّر، ويتعلق بالتابعَين `fs.fchown()` و `fs.lchown()`.
- `fs.close()`: يغلق واصف الملف file descriptor.

- `fs.copyFile()`: ينسخ ملفًا.
- `fs.createReadStream()`: ينشئ مجرى stream ملف قابل للقراءة.
- `fs.createWriteStream()`: ينشئ مجرى ملف قابل للكتابة.
- `fs.link()`: ينشئ رابطًا صلبًا hard link جديدًا إلى ملف.
- `fs.mkdir()`: ينشئ مجلدًا جديدًا.
- `fs.mkdtemp()`: ينشئ مجلدًا مؤقتًا.
- `fs.open()`: يضبط نمط الملف.
- `fs.readdir()`: يقرأ محتويات مجلد.
- `fs.readFile()`: يقرأ محتوى ملف، ويتعلق بالتابع `fs.read()`.
- `fs.readlink()`: يقرأ قيمة الوصلة الرمزية symbolic link.
- `fs.realpath()`: يُستخدَم لربط resolve مؤشرات مسار الملف النسبي (`.` و `..`) مع المسار الكامل.
- `fs.rename()`: يعيد تسمية ملف أو مجلد.
- `fs.rmdir()`: يزيل مجلدًا.
- `fs.stat()`: يعيد حالة الملف المحدَّد بواسطة اسم الملف المُمرَّر، ويتعلق بالتابعَين `fs.fstat()` و `fs.lstat()`.
- `fs.symlink()`: ينشئ وصلةً رمزيةً جديدًا إلى ملف.
- `fs.truncate()`: يقتطع الملف المحدَّد بواسطة اسم الملف المُمرَّر إلى طول معيَّن، ويتعلق بالتابع `fs.ftruncate()`.
- `fs.unlink()`: يزيل ملفًا أو وصلةً رمزيةً.
- `fs.unwatchFile()`: يوقِف مشاهدة التغييرات على ملف.
- `fs.utimes()`: يغيّر الطابع الزمني timestamp للملف المحدَّد باسم الملف المُمرَّر، ويتعلق بالتابع `fs.futimes()`.
- `fs.watchFile()`: يبدأ بمشاهدة التغييرات على ملف، ويتعلق بالتابع `fs.watch()`.
- `fs.writeFile()`: يكتب بيانات في ملف، ويتعلق بالتابع: `fs.write()`.

جميع التوابع في وحدة `fs` غير متزامنة افتراضيًا، ولكن يمكنها العمل بطريقة متزامنة من خلال إلحاق الكلمة `Sync` باسم التابع كما يلي:

- `fs.rename()`
- `fs.renameSync()`
- `fs.write()`
- `fs.writeSync()`

ويحدِث ذلك فرقًا كبيرًا في تدفق تطبيقك.

> تتضمن نود 10 أنواع من الدعم التجريبي لواجهة برمجة تطبيقات قائمة على الوعود promise.

لنختبر التابع `fs.rename()` مثلًا، حيث تُستخدَم واجهة برمجة التطبيقات API غير المتزامنة مع دالة رد نداء callback:

```
const fs = require('fs')
fs.rename('before.json', 'after.json', (err) => {
  if (err) {
    return console.error(err)
  }
  //done
})
```

يمكن استخدام واجهة برمجة تطبيقات متزامنة مثل المثال التالي مع كتلة try/catch لمعالجة الأخطاء:

```
const fs = require('fs')
try {
  fs.renameSync('before.json', 'after.json')
  //done
} catch (err) {
  console.error(err)
}
```

الاختلاف الرئيسي هو إيقاف تنفيذ السكربت الخاص بك في المثال الثاني إلى أن تنجح عملية الملف.

للمزيد من المعلومات حول هذه الوحدة، يمكنك الرجوع إلى توثيق التعامل مع نظام الملفات في Node.js في موسوعة حسوب.

8.2 وحدة المسار path

توفِّر وحدة path عمليات متعددة ومفيدة للوصول إلى نظام الملفات والتفاعل معه، وليست هناك حاجة لتثبيتها نظرًا لكونها جزءًا من نواة نود، إذ يمكن استخدامها ببساطة عن طريق طلبها كما يلي:

```
const path = require('path')
```

توفِّر هذه الوحدة محرف path.sep الذي يوفّر فاصل مقاطع المسار path segment separator وهو (\ على نظام ويندوز Windows و/ على نظامي لينكس Linux وmacOS)، بالإضافة إلى محرف path.delimiter الذي يوفّر محدّد المسار path delimiter وهو (; على ويندوز Windows و : على نظامَي لينكس Linux و macOS).

توابع وحدة path هي:

- path.basename()
- path.dirname()
- path.extname()
- path.isAbsolute()
- path.join()
- path.normalize()
- path.parse()
- path.relative()
- path.resolve()

للمزيد من المعلومات حول هذه الوحدة، يمكنك الرجوع إلى توثيق وحدة المسار (Path) في Node.js في موسوعة حسوب.

8.2.1 التابع path.basename()

يعيد هذا التابع الجزء الأخير من المسار، ويمكن للمعامِل الثاني تحديد امتداد الملف لإعطاء الملف دون امتداده كما يلي:

```
require('path').basename('/test/something') //something
require('path').basename('/test/something.txt') //something.txt
require('path').basename('/test/something.txt', '.txt') //something
```

8.2.2 التابع path.dirname()

يعيد هذا التابع جزء المجلد أو الدليل من المسار كما يلي:

```
require('path').dirname('/test/something') // /test
require('path').dirname('/test/something/file.txt') // /test/something
```

8.2.3 التابع path.extname()

يعيد هذا التابع جزء الامتداد من المسار كما يلي:

```
require('path').dirname('/test/something') // ''
require('path').dirname('/test/something/file.txt') // '.txt'
```

8.2.4 التابع path.isAbsolute()

يعيد هذا التابع القيمة `true` إذا كان المسار مسارًا مطلقًا.

```
require('path').isAbsolute('/test/something') // true
require('path').isAbsolute('./test/something') // false
```

8.2.5 التابع path.join()

يربط هذا التابع جزأين أو أكثر من المسار مع بعضها البعض كما يلي:

```
const name = 'flavio'
require('path').join('/', 'users', name, 'notes.txt')
//'/users/flavio/notes.txt'
```

8.2.6 التابع path.normalize()

يحاول هذا التابع حساب المسار الفعلي عندما يحتوي على محددات نسبية مثل `.` أو `..` أو شرطات مائلة مزدوجة:

```
require('path').normalize('/users/flavio/..//test.txt')
///users/test.txt
```

8.2.7 التابع path.parse()

يوزّع هذا التابع مسارًا على كائن يتكون من أجزاء متعددة هي:

- `root`: يمثّل الجذر.

- `dir`: هو مسار المجلد بداية من الجذر.
- `base`: يمثّل اسم الملف مع الامتداد.
- `name`: هو اسم الملف.
- `ext`: يمثّل امتداد الملف.

إليك المثال التالي:

```
require('path').parse('/users/test.txt')
```

وتكون النتيجة كما يلي:

```
{
  root: '/',
  dir: '/users',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
}
```

8.2.8 التابع path.relative()

يقبل هذا التابع مسارين على أساس وسائط، ويعيد المسار النسبي من المسار الأول إلى المسار الثاني بناءً على مجلد العمل الحالي مثل المثال التالي:

```
require('path').relative('/Users/flavio', '/Users/flavio/test.txt')
//'test.txt'
require('path').relative('/Users/flavio',
'/Users/flavio/something/test.txt') //'something/test.txt'
```

8.2.9 التابع path.resolve()

يمكنك حساب المسار المطلق لمسار نسبي باستخدام التابع `path.resolve()` كما يلي:

```
path.resolve('flavio.txt') // '/Users/flavio/flavio.txt' إذا شُغِّل من
المجلد المحلي
```

إذا حدّدت المعامل الثاني، فسيستخدم التابع `resolve` المعامِل الأول أساسًا للمعامِل الثاني كما يلي:

```
path.resolve('tmp', 'flavio.txt') // '/Users/flavio/tmp/flavio.txt' إذا
شُغِّل من المجلد المحلي
```

إذا بدأ المعامِل الأول بشرطة مائلة، فهذا يعني أنه مسار مطلق كما يلي:

```
path.resolve('/etc', 'flavio.txt')//'/etc/flavio.txt'
```

8.3 وحدة os

توفِّر هذه الوحدة عمليات متعددة يمكنك استخدامها لاسترداد معلومات من نظام التشغيل الأساسي والحاسوب الذي يعمل عليه البرنامج والتفاعل معه.

```
const os = require('os')
```

هناك بعض الخاصيات المفيدة التي تخبرنا ببعض الأمور الأساسية المتعلقة بمعالجة الملفات مثل:

- `os.EOL` التي تعطينا متسلسلة محدّد السطور، وهي `\n` على نظامَي لينكس Linux وmacOS؛ أما على نظام ويندوز Windows فهي `\r\n`.

> نقصد بنظامَي لينكس و macOS منصات POSIX، واستبعدنا بهدف التبسيط أنظمة التشغيل الأخرى الأقل شيوعًا التي يمكن تشغيل نود Node عليها.

- `os.constants.signals` التي تعطينا كل الثوابت المتعلقة بمعالجة إشارات العمليات مثل SIGHUP وSIGKILL وما إلى ذلك، كما يمكنك الاطلاع على جميع هذه الثوابت على node_os.
- `os.constants.errno` التي تضبط الثوابت في تقارير الخطأ مثل EADDRINUSE و EOVERFLOW وغير ذلك.

لنتعرّف الآن على التوابع الرئيسية التي توفرها وحدة `os` وهي:

- `os.arch()`
- `os.cpus()`
- `os.endianness()`
- `os.freemem()`
- `os.homedir()`
- `os.hostname()`
- `os.hostname()`
- `os.loadavg()`
- `os.networkInterfaces()`
- `os.platform()`

- `os.release()`
- `os.tmpdir()`
- `os.totalmem()`
- `os.type()`
- `os.uptime()`
- `os.userInfo()`

للمزيد من المعلومات حول هذه الوحدة، يمكنك الرجوع إلى توثيق الوحدة os في Node.js في موسوعة حسوب.

8.3.1 التابع ()os.arch

يعيد هذا التابع السلسلة النصية التي تحدد البنية الأساسية مثل `arm` و `x64` و `arm64`.

8.3.2 التابع ()os.cpus

يعيد معلومات وحدات المعالجة المركزية المتوفرة على نظامك، كالمعلومات التالية على سبيل المثال:

```
[ { model: 'Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz',
    speed: 2400,
    times:
    { user: 281685380,
      nice: 0,
      sys: 187986530,
      idle: 685833750,
      irq: 0 } },
  { model: 'Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz',
    speed: 2400,
    times:
    { user: 282348700,
      nice: 0,
      sys: 161800480,
      idle: 703509470,
      irq: 0 } } ]
```

8.3.3 التابع ()os.endianness

يعيد هذا التابع القيمة BE أو القيمة LE بناءً على طريقة تصريف نود باستخدام تخزين البتات الأقل أهمية أولًا Big Endian أو تخزين البتات الأكثر أهمية أولًا Little Endian.

8.3.4 التابع ()os.freemem

يعيد هذا التابع عدد البايتات التي تمثل الذاكرة المتاحة في النظام.

8.3.5 التابع ()os.homedir

يعيد هذا التابع المسار إلى مجلد المستخدِم الحالي الرئيسي مثل المثال التالي:

```
'/Users/flavio'
```

8.3.6 التابع ()os.hostname

يعيد هذا التابع اسم المضيف hostname.

8.3.7 التابع ()os.loadavg

يعيد هذا التابع الحساب الذي أجراه نظام التشغيل على متوسط التحميل، حيث يعيد فقط قيمة ذات معنى في نظامَي لينكس Linux وmacOS مثل المثال التالي:

```
[ 3.68798828125, 4.00244140625, 11.1181640625 ]
```

8.3.8 التابع ()os.networkInterfaces

يعيد هذا التابع تفاصيل واجهات الشبكة المتوفرة على نظامك، وإليك المثال التالي:

```
{ lo0:
 [ { address: '127.0.0.1',
     netmask: '255.0.0.0',
     family: 'IPv4',
     mac: 'fe:82:00:00:00:00',
     internal: true },
   { address: '::1',
     netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
     family: 'IPv6',
     mac: 'fe:82:00:00:00:00',
```

```
      scopeid: 0,
      internal: true },
    { address: 'fe80::1',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: 'fe:82:00:00:00:00',
      scopeid: 1,
      internal: true } ],
  en1:
  [ { address: 'fe82::9b:8282:d7e6:496e',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: '06:00:00:02:0e:00',
      scopeid: 5,
      internal: false },
  { address: '192.168.1.38',
      netmask: '255.255.255.0',
      family: 'IPv4',
      mac: '06:00:00:02:0e:00',
      internal: false } ],
  utun0:
  [ { address: 'fe80::2513:72bc:f405:61d0',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: 'fe:80:00:20:00:00',
      scopeid: 8,
      internal: false } ] }
```

8.3.9 التابع ()os.platform

يعيد هذا التابع المنصة الذي جرى تصريف نود من أجلها مثل:

- `darwin`
- `freebsd`
- `linux`
- `openbsd`

- `win32`
- وغيرها الكثير.

8.3.10 التابع ()os.release

يعيد هذا التابع سلسلةً نصيةً تحدّد رقم إصدار نظام التشغيل.

8.3.11 التابع ()os.tmpdir

يعيد هذا التابع المسار إلى المجلد المؤقت المعيَّن.

8.3.12 التابع ()os.totalmem

يعيد هذا التابع عدد البايتات الذي يمثِّل إجمالي الذاكرة المتوفرة في النظام.

8.3.13 التابع ()os.type

يحدّد هذا التابع نظام التشغيل كما يلي:

- `Linux`
- `Darwin` على نظام macOS.
- `Windows_NT` على نظام ويندوز.

8.3.14 التابع ()os.uptime

يعيد هذا التابع عدد الثواني التي عمل فيها الحاسوب منذ آخر إعادة تشغيل.

8.3.15 التابع ()os.userInfo

يعيد معلومات عن المستخدِم الفعّال حاليًا.

8.4 وحدة الأحداث events

توفِّر لنا وحدة الأحداث `events` الصنف EventEmitter، وتُعَدّ أساسًا للعمل مع الأحداث في نود.

```
const EventEmitter = require('events')
const door = new EventEmitter()
```

يختبر مستمع الأحداث event listener أحداثه الخاصة ويستخدِم الحدثين التاليين:

- `newListener` عند إضافة المستمع.

- `removeListener` عند إزالة المستمع.

سنشرح فيما يلي التوابع المفيدة التالية:

- `emitter.addListener()`
- `emitter.emit()`
- `emitter.eventNames()`
- `emitter.getMaxListeners()`
- `emitter.listenerCount()`
- `emitter.listeners()`
- `emitter.off()`
- `emitter.on()`
- `emitter.once()`
- `emitter.prependListener()`
- `emitter.prependOnceListener()`
- `emitter.removeAllListeners()`
- `emitter.removeListener()`
- `emitter.setMaxListeners()`

للمزيد من المعلومات حول هذه الوحدة، يمكنك الرجوع إلى توثيق الأحداث في Node.js في موسوعة حسوب.

8.4.1 التابع emitter.addListener()

وهو الاسم البديل للتابع `emitter.on()`.

8.4.2 التابع emitter.emit()

يصدر هذا التابع حدثًا، حيث يستدعي بصورة متزامنة كل مستمع حدث بالترتيب الذي سُجِّلت به.

8.4.3 التابع emitter.eventNames()

يعيد هذا التابع مصفوفةً من السلاسل النصية التي تمثِّل الأحداث المُسجَّلة في كائن EventListener الحالي:

```
door.eventNames()
```

8.4.4 التابع ()emitter.getMaxListeners

يُستخدَم هذا التابع للحصول على الحد الأقصى من المستمعين الذي يمكن إضافته إلى كائن EventListener، حيث يُضبَط هذا العدد افتراضيًا على القيمة 10 ولكن يمكن زيادته أو إنقاصه باستخدام `setMaxListeners()`.

```
door.getMaxListeners()
```

8.4.5 التابع ()emitter.listenerCount

يُستخدَم هذا التابع للحصول على عدد مستمعي الحدث المُمرَّرين على أساس معاملات كما يلي:

```
door.listenerCount('open')
```

8.4.6 التابع ()emitter.listeners

يُستخدَم هذا التابع للحصول على مصفوفة مستمعي الحدث المُمرَّرين على أساس معاملات كما يلي:

```
door.listeners('open')
```

8.4.7 التابع ()emitter.off

يمثّل هذا التابع الاسم البديل للتابع `emitter.removeListener()` المُضاف في الإصدار 10 من نود.

8.4.8 التابع ()emitter.on

يضيف هذا التابع دالة رد النداء التي تُستدعَى عند إصدار حدث، ويُستخدَم هذا التابع كما يلي:

```
door.on('open', () => {
  console.log('Door was opened')
})
```

8.4.9 التابع ()emitter.once

يضيف هذا التابع دالة رد النداء التي تُستدعَى عند إصدار حدث لأول مرة بعد تسجيله، حيث ستُستدعَى دالة رد النداء تلك مرةً واحدةً فقط، ولن تُستدعَى مرةً أخرى.

لاحظ المثال التالي:

```
const EventEmitter = require('events')
const ee = new EventEmitter()
```

```
ee.once('my-event', () => {
  //استدعِ دالة رد النداء مرةً واحدةً
})
```

8.4.10 التابع ()emitter.prependListener

يُضاف المستمع الذي تضيفه باستخدام `on` أو `addListener` في آخر طابور المستمعين ويُستدعَى أخيرًا كذلك، ولكنه يُضاف ويُستدعَى قبل المستمعين الآخرين باستخدام `prependListener`.

8.4.11 التابع ()emitter.prependOnceListener

يُضاف المستمع الذي تضيفه باستخدام `once` في آخر طابور المستمعين ويُستدعَى أخيرًا كذلك، ولكنه يُضاف ويُستدعَى قبل المستمعين الآخرين باستخدام `prependOnceListener`.

8.4.12 التابع ()emitter.removeAllListeners

يزيل هذا التابع جميع مستمعي الكائن الذي يصدر الأحداث ويستمع إلى حدث محدَّد:

```
door.removeAllListeners('open')
```

8.4.13 التابع ()emitter.removeListener

يزيل مستمعًا محدَّدًا عن طريق حفظ دالة رد النداء في متغير عند إضافته، بحيث يمكنك الإشارة إليه لاحقًا:

```
const doSomething = () => {}
door.on('open', doSomething)
door.removeListener('open', doSomething)
```

8.4.14 التابع ()emitter.setMaxListeners

يضبط الحد الأقصى لعدد المستمعين الذي يمكن إضافته إلى كائن EventListener، حيث يُضبَط هذا العدد افتراضيًا على القيمة 10 ولكن يمكن زيادته أو إنقاصه.

```
door.setMaxListeners(50)
```

8.5 وحدة HTTP

توفّر وحدة http في Node.js دوالًا وأصنافًا مفيدة لبناء خادم HTTP، وتُعَدّ الوحدة الأساسية لشبكات نود، كما يمكن تضمين وحدة http كما يلي:

```
const http = require('http')
```

توفّر وحدة http بعض الخاصيات properties والتوابع methods والأصناف classes.

للمزيد من المعلومات حول هذه الوحدة، يمكنك الرجوع إلى توثيق الوحدة HTTP في Node.js في موسوعة حسوب.

8.5.1 الخاصيات

توفِّر وحدة HTTP الخاصيات التالية:

- `http.METHODS`
- `http.STATUS_CODES`
- `http.globalAgent`

ا. الخاصية http.METHODS

تعطي هذه الخاصية قائمةً بجميع توابع HTTP المدعومة كما يلي:

```
> require('http').METHODS
 [ 'ACL',
   'BIND',
   'CHECKOUT',
   'CONNECT',
   'COPY',
   'DELETE',
   'GET',
   'HEAD',
   'LINK',
   'LOCK',
   'M-SEARCH',
   'MERGE',
   'MKACTIVITY',
   'MKCALENDAR',
   'MKCOL',
   'MOVE',
   'NOTIFY',
   'OPTIONS',
   'PATCH',
```

```
    'POST',
    'PROPFIND',
    'PROPPATCH',
    'PURGE',
    'PUT',
    'REBIND',
    'REPORT',
    'SEARCH',
    'SUBSCRIBE',
    'TRACE',
    'UNBIND',
    'UNLINK',
    'UNLOCK',
    'UNSUBSCRIBE' ]
```

ب. الخاصية http.STATUS_CODES

تعطي هذه الخاصية قائمةً بجميع رموز حالة HTTP ووصفها كما يلي:

```
> require('http').STATUS_CODES
 { '100': 'Continue',
   '101': 'Switching Protocols',
   '102': 'Processing',
   '200': 'OK',
   '201': 'Created',
   '202': 'Accepted',
   '203': 'Non-Authoritative Information',
   '204': 'No Content',
   '205': 'Reset Content',
   '206': 'Partial Content',
   '207': 'Multi-Status',
   '208': 'Already Reported',
   '226': 'IM Used',
   '300': 'Multiple Choices',
   '301': 'Moved Permanently',
   '302': 'Found',
   '303': 'See Other',
```

```
    '304': 'Not Modified',
    '305': 'Use Proxy',
    '307': 'Temporary Redirect',
    '308': 'Permanent Redirect',
    '400': 'Bad Request',
    '401': 'Unauthorized',
    '402': 'Payment Required',
    '403': 'Forbidden',
    '404': 'Not Found',
    '405': 'Method Not Allowed',
    '406': 'Not Acceptable',
    '407': 'Proxy Authentication Required',
    '408': 'Request Timeout',
    '409': 'Conflict',
    '410': 'Gone',
    '411': 'Length Required',
    '412': 'Precondition Failed',
    '413': 'Payload Too Large',
    '414': 'URI Too Long',
    '415': 'Unsupported Media Type',
    '416': 'Range Not Satisfiable',
    '417': 'Expectation Failed',
    '418': 'I\'m a teapot',
    '421': 'Misdirected Request',
    '422': 'Unprocessable Entity',
    '423': 'Locked',
    '424': 'Failed Dependency',
    '425': 'Unordered Collection',
    '426': 'Upgrade Required',
    '428': 'Precondition Required',
    '429': 'Too Many Requests',
    '431': 'Request Header Fields Too Large',
    '451': 'Unavailable For Legal Reasons',
    '500': 'Internal Server Error',
    '501': 'Not Implemented',
    '502': 'Bad Gateway',
```

```
    '503': 'Service Unavailable',
    '504': 'Gateway Timeout',
    '505': 'HTTP Version Not Supported',
    '506': 'Variant Also Negotiates',
    '507': 'Insufficient Storage',
    '508': 'Loop Detected',
    '509': 'Bandwidth Limit Exceeded',
    '510': 'Not Extended',
    '511': 'Network Authentication Required' }
```

ج. الخاصية http.globalAgent

تؤشّر هذه الخاصية إلى نسخة كائن الوكيل Agent العامة، والذي هو نسخة من الصنف `http.Agent`، حيث يُستخدَم هذا الصنف لإدارة الاتصالات المستمرة وإعادة استخدام عملاء HTTP، وهو المكوّن الأساسي من شبكات HTTP الخاصة بنود Node.

8.5.2 التوابع

توفِّر وحدة HTTP التوابع التالية:

- `http.createServer()`
- `http.request()`
- `http.get()`

ا. التابع http.createServer()

يعيد هذا التابع نسخةً جديدةً من الصنف `http.Server`، حيث يُستخدَم كما يلي:

```
const server = http.createServer((req, res) => {
  //معالجة كل طلب باستخدام دالة رد النداء هذه
})
```

ب. التابع http.request()

ينشئ طلب HTTP إلى خادم، مما يؤدي إلى إنشاء نسخة من الصنف `http.ClientRequest`.

ج. التابع http.get()

يشبه التابع `http.request()`، ولكنه يضبط تلقائيًا تابع HTTP على GET ويستدعي التابع `req.end()` تلقائيًا.

8.5.3 الأصناف Classes

توفِّر وحدة HTTP خمسة أصناف هي:

- `http.Agent`
- `http.ClientRequest`
- `http.Server`
- `http.ServerResponse`
- `http.IncomingMessage`

ا. الصنف http.Agent

تُنشئ نود نسخةً عامةً من الصنف `http.Agent` لإدارة الاتصالات المستمرة وإعادة استخدام عملاء HTTP، وهو المكوّن الأساسي من شبكات HTTP الخاصة بنود Node، كما يتأكّد هذا الكائن من وضع كل طلب إلى الخادم في طابور ومن إعادة استخدام المقبس socket، كما أنه يحتفظ بمجمّع من المقابس بهدف تحسين الأداء.

ب. الصنف http.ClientRequest

يُنشَأ الكائن `http.ClientRequest` عند استدعاء التابع `http.request()` أو التابع `http.get()`، حيث يُستدعَى حدث `response` مع الاستجابة عند تلقيها باستخدام نسخة من كائن `http.IncomingMessage` على أساس وسيط، ويمكن قراءة بيانات الاستجابة المُعادة بطريقتين هما:

- استدعاء التابع `response.read()`.
- يمكنك إعداد مستمعٍ للحدث `data` في معالج الحدث `response` بحيث يمكنك الاستماع للبيانات المتدفقة إليه.

ج. الصنف http.Server

تُنشَأ وتُعاد عادةً نسخة من هذا الصنف عند إنشاء خادم جديد باستخدام التابع `http.createServer()`، يمكنك الوصول إلى توابع كائن خادم بعد إنشائه، وهذه التوابع هي:

- `close()` الذي يوقِف الخادم من قبول اتصالات جديدة.
- `listen()` الذي يشغّل خادم HTTP ويستمع للاتصالات.

د. الصنف http.ServerResponse

ينشئه الصنف `http.Server` ويمرّره على أساس معامل ثانٍ لحدث `request` الذي يشغّله، حيث يُعرَف هذا الصنف ويُستخدَم في الشيفرة على أنه كائن `res` كما يلي:

```
const server = http.createServer((req, res) => {
  //http.ServerResponse كائن هو res
})
```

التابع الذي ستستدعيه دائمًا في المعالج هو `end()` والذي يغلق الاستجابة بعد اكتمال الرسالة ثم يستطيع الخادم إرسالها إلى العميل، إذ يجب استدعاؤه في كل استجابة، وتُستخدَم التوابع التالية للتفاعل مع ترويسات HTTP:

- `getHeaderNames()`: للحصول على قائمة بأسماء ترويسات HTTP المضبوطة مسبقًا.
- `getHeaders()`: للحصول على نسخة من ترويسات HTTP المضبوطة مسبقًا.
- `setHeader('headername', value)`: يحدّد قيمة ترويسة HTTP.
- `getHeader('headername')`: للحصول على ترويسة HTTP المضبوطة مسبقًا.
- `removeHeader('headername')`: يزيل ترويسة HTTP المضبوطة مسبقًا.
- `hasHeader('headername')`: يعيد القيمة true إذا احتوت الاستجابة على هذه الترويسة المضبوطة.
- `headersSent()`: يعيد القيمة true إذا أُرسِلت الترويسات إلى العميل مسبقًا.

يمكنك إرسال الترويسات بعد معالجتها إلى العميل عن طريق استدعاء التابع `response.writeHead()` الذي يقبل رمز الحالة statusCode على أساس معامل أول، ورسالة الحالة الاختيارية، وكائن الترويسات، كما يمكنك إرسال البيانات إلى العميل في جسم الاستجابة عن طريق استخدام التابع `write()` الذي سيرسل البيانات المخزَّنة إلى مجرى استجابة HTTP، فإذا لم تُرسَل الترويسات بعد باستخدام التابع `response.writeHead()`، فستُرسَل الترويسات أولًا مع رمز الحالة والرسالة المحدَّدة في الطلب والتي يمكنك تعديلها عن طريق ضبط قيم الخاصيات `statusCode` و`statusMessage` كما يلي:

```
response.statusCode = 500
response.statusMessage = 'Internal Server Error'
```

ه. الصنف http.IncomingMessage

يُنشَأ كائن `http.IncomingMessage` باستخدام:

- `http.Server` عند الاستماع إلى الحدث `request`.
- `http.ClientRequest` عند الاستماع إلى الحدث `response`.

يمكن استخدام كائن `http.IncomingMessage` للوصول إلى خاصيات الاستجابة التالية:

- الحالة status باستخدام توابع `statusCode` و`statusMessage` الخاصة به.
- الترويسات باستخدام توابع `headers` أو `rawHeaders` الخاصة به.
- تابع HTTP باستخدام تابع `method` الخاص به.
- إصدار HTTP باستخدام تابع `httpVersion`.
- عنوان URL باستخدام تابع `url`.
- المقبس الأساسي underlying socket باستخدام تابع `socket`.

يمكن الوصول إلى البيانات باستخدام المجاري streams، حيث ينفّذ كائن `http.IncomingMessage` واجهة المجرى Stream القابلة للقراءة.

8.6 وحدة MySQL

تُعَدّ MySQL واحدةً من أكثر قواعد البيانات العلائقية شيوعًا في العالم، إذ يحتوي نظام نود Node المجتمعي على حزم مختلفة تتيح لك التعامل مع MySQL وتخزين البيانات واسترداد البيانات وما إلى ذلك، كما سنستخدِم حزمة mysqljs/mysql التي تحتوي على أكثر من 12000 نجمة على GitHub وهي موجودة منذ سنوات.

8.6.1 تثبيت حزمة نود mysql

يمكنك تثبيتها باستخدام الأمر التالي:

```
npm install mysql
```

8.6.2 تهيئة الاتصال بقاعدة البيانات

يجب تضمين الحزمة أولًا كما يلي:

```
const mysql = require('mysql')
```

ثم تنشئ اتصالًا كما يلي:

```
const options = {
  user: 'the_mysql_user_name',
  password: 'the_mysql_user_password',
  database: 'the_mysql_database_name'
}
  const connection = mysql.createConnection(options)
```

ثم تهيئ اتصالًا جديدًا عن طريق استدعاء ما يلي:

```
connection.connect(err => {
  if (err) {
    console.error('An error occurred while connecting to the DB')
    throw err
  }
})
```

8.6.3 خيارات الاتصال

احتوى كائن options في المثال السابق على 3 خيارات هي:

```
const options = {
  user: 'the_mysql_user_name',
  password: 'the_mysql_user_password',
  database: 'the_mysql_database_name'
}
```

هناك خيارات أخرى متعددة يمكنك استخدامها مثل:

- host: اسم مضيف قاعدة البيانات، وقيمته الافتراضية هي localhost.
- port رقم منفذ خادم MySQL، وقيمته الافتراضية هي 3306.
- socketPath: يُستخدَم لتحديد مقبس يونيكس unix بدلًا من host و port.
- debug: يمكن استخدامه لتنقيح الأخطاء debugging عند تعطيله افتراضيًا.
- trace: يطبع تعقبات المكدس stack traces عند حدوث الأخطاء عند تفعيله افتراضيًا.
- ssl: يُستخدَم لإعداد اتصال SSL إلى الخادم.

8.6.4 إجراء استعلام SELECT

أصبحتَ الآن جاهزًا لإجراء استعلام SQL في قاعدة البيانات، وسيستدعي الاستعلامُ بمجرد تنفيذه دالةَ رد النداء التي تحتوي على الخطأ المُحتمَل error والنتائج results والحقول fields كما يلي:

```
connection.query('SELECT * FROM todos', (error, todos, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
    console.log(todos)
 })
```

يمكنك تمرير القيم التي ستُتجاوز تلقائيًا كما يلي:

```
const id = 223
connection.query('SELECT * FROM todos WHERE id = ?', [id], (error,
todos, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
  console.log(todos)
})
```

يمكنك تمرير قيم متعددة من خلال وضع مزيد من العناصر في المصفوفة التي تمررها على أساس معامل ثانٍ كما يلي:

```
const id = 223
const author = 'Flavio'
connection.query('SELECT * FROM todos WHERE id = ? AND author = ?',
[id, author], (error,
todos, fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
  console.log(todos)
})
```

8.6.5 إجراء استعلام INSERT

يمكنك تمرير كائن كما يلي:

```
const todo = {
  thing: 'Buy the milk'
  author: 'Flavio'
}
connection.query('INSERT INTO todos SET ?', todo, (error, results,
fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }
})
```

إذا احتوى الجدول على مفتاح رئيسي primary key مع `auto_increment`، فستُعاد قيمته ضمن القيمة `results.insertId` كما يلي:

```
const todo = {
  thing: 'Buy the milk'
  author: 'Flavio'
}
connection.query('INSERT INTO todos SET ?', todo, (error, results,
fields) => {
  if (error) {
    console.error('An error occurred while executing the query')
    throw error
  }}
  const id = results.resultId
  console.log(id)
)
```

8.6.6 إغلاق الاتصال

يمكنك استدعاء التابع `end()` عند إنهاء الاتصال بقاعدة البيانات كما يلي:

```
connection.end()
```

يعمل ذلك على التأكد من إرسال أي استعلام مُعلَّق وإنهاء الاتصال بأمان.

8.7 وحدات مخصصة

إذا لم تعثر على الوحدات المناسبة لـك ضـمن الوحـدات الأساسـية، فيمكنـك بنـاء وحـدة مخصَّصـة تخـدم غرضك بالاعتماد على الوحدات الأساسية ويمكنك أن تصدِّرها وتستوردها حـتى أنـه يمكنـك بنـاء مكتبـة كاملـة، حيث سنتعرّف فيما يلي على كيفية استخدام واجهة module.exports البرمجيـة لتصـدير بياناتـك إلى ملفـات أخرى في تطبيقك أو إلى تطبيقات أخرى.

يمتلك نود نظام وحدات مبنيّ مسبقًا، إذ يمكن لملف Node.js اسـتيراد العمليـات الـتي تصـدّرها ملفـات Node.js الأخـرى، فـإذا أردت اسـتيراد شـيءٍ مـا، فاسـتخدم مـا يلي لاسـتيراد العمليـات الظـاهرة في ملـف `library.js` الموجود في مجلد الملف الحالي، إذ يجب إظهار العمليـات في هـذا الملـف قبـل أن تسـتوردها ملفات أخرى:

```
const library = require('./library')
```

يكون أيّ كائن أو متغير آخر مُعرَّف في الملف خاصًا private افتراضيًا ولا يظهر لأيّ شيء خارجي، وهـذا مـا تسمح لنا به واجهة برمجة تطبيقات `module.exports` التي يوفِّرها نظام module، وإذا أسندتَ كائنًا أو دالـةً مثل خاصية `exports` جديدة، فهذا هو الشيء الذي يظهر، ويمكن استيراده على هذا النحو في أجـزاء أخـرى من تطبيقك أو في تطبيقـات أخـرى أيضًـا، حيث يمكنـك تطبيق ذلـك بطريقـتين، الأولى هي إسـناد كـائن لوحـدة `module.exports`، وهو كائن خارجي يوفّره نظام module، وبالتالي سيصدّر ملفك هذا الكائن فقط:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}
module.exports = car
//..في الملف الآخر
const car = require('./car')
```

أما الطريقة الثانية فهي إضافة الكائن المُصدَّر على أساس خاصية `exports`، حيث تتيح لك هـذه الطريقـة تصدير كائنات أو دوال أو بيانات متعددة:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}
exports.car = car
```

أو مباشرةً كما يلي:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}
```

كما ستستخدمه في الملف الآخر من خلال الإشارة إلى خاصية الاستيراد كما يلي:

```
const items = require('./items')
items.car
```

أو كما يلي:

```
const car = require('./items').car
```

هناك فرق بين `module.exports` و `exports`، فالأول يُظهِر الكائن الذي يؤشّر إليه، بينما يُظهِر الثاني خاصيات الكائن الذي يؤشّر إليه.

تعرفت في هذا الفصل على تعرف على وحدات Node.js الأساسية، كما توفر Node.js عددًا هائلًا من الوحدات والمكتبات الأخرى التي يمكنك الاستفادة منها لتعزز تطوير التطبيقات وتسهل عليك عملية البرمجة وتطويرها.

8.8 الخاتمة

إلى هنا نكون قد وصلنا لختام كتابنا الذي شرحنا فيه أساسيات بيئة Node.js لتشغيل جافا سكريبت على الخادم. وتعرفنا على أبرز مميزاتها ودورها في تطوير الويب وتعرفنا كيف أتاحت للمطورين استخدام لغة جافاسكريبت على الجانبين الأمامي والخلفي من التطبيق، ومكنت مطوري الواجهة الأمامية استخدام نفس اللغة دون الحاجة إلى تعلم لغة جديدة بالكامل، ابدأ بتطوير تطبيقاتك باستخدامها واستفد من إمكانياتها المميزة.

مستقل
mostaql.com

ادخل سوق العمل ونفذ المشاريع باحترافية
عبر أكبر منصة عمل حر بالعالم العربي

ابدأ الآن كمستقل

أحدث إصدارات أكاديمية حسوب