NODE JS

عربي

المحتويات باختصار

جدول المحتويات

تمهيد

تعـد Node.js بيئـة تشـغيل مفتوحـة المصـدر يمكن خلالهـا تنفيـذ شـيفرات مكتوبـة بلغـة جافاسـكربت JavaScript دون الحاجة إلى متصفح ويب، وبـذلك لم يعـد المتصـفح المشـغل الوحيـد والحصـري لهـا مـا فتح الآفاق لاستخدام جافاسـكربت في مختلـف المجـالات وليس فقـط في تطـوير الواجهـات الأماميـة front-end لصفحات ومواقع الويب وإضافة الفاعلية عليها مع لغة HTML ولغة CSS، وتُستعمل Node.js عـادةً في تطـوير الواجهات الخلفية لتطبيقات ومواقع الـويب عـبر بنـاء خـوادم ويب خلفيـة كمـا يمكن اسـتعمالها لتطـوير أدوات وبرامج تعمل من سطر الأوامر.

حول الكتاب

ستتعلم في هذا الكتاب أساسيات البرمجة باستخدام Node.js بأسلوب عملي تطبيقي إذ ستبني خلال هذا الكتاب عدة تطبيقات وخوادم ويب مختلفة، وستصبح قادرًا في نهايته على كتابة برامج تستخدم مختلف مـيزات Node.js منها التنفيذ الغير متزامن والتعامل مع الأحداث والتحكم بالعمليات وإدارة الوحدات.

كما سيتطرق الكتـاب إلى بعض المواضـيع المتقدمـة في البرمجـة منهـا كيفيـة تنقيح تطبيقـات Node.js وتصحيح الأخطاء فيها باستخدام الأدوات المتـوفرة سـواءً في نـود نفسـها أو في المتصـفح عـبر أدوات التطـوير DevTools، وأيضًا كيفية كتابة وحدات اختبار unite tests لوظائف التطبيق للتأكد من عملها وفق المطلوب.

هذا الكتاب مُترجم عن كتاب How To Code in Node.js بواسطة موقع DigitalOcean وقد ساهم بـه مجموعة مؤلفين من فريق Stack Abuse

1. كتابة أول برنامج وتنفيذه

Node.js -تلفظ نود جي إس- هو بيئة تشغيل جافاسكربت مفتوحة المصدر تتيح تنفيذ شيفرات جافاسكربت خارج المتصفح، وذلك باستخدام محرك جافاسكربت V8 الشهير المُستخدم ضمن متصفح جوجل كروم، ومن أشهر استخدامات هذه البيئة هو تطوير تطبيقات وخوادم الويب وحتى أدوات سطر الأوامر، وتوفر لنا هذه البيئة كتابة شيفرات الواجهات الأمامية Front-end والواجهات الخلفية Back-end بلغة برمجة واحدة وهي جافاسكربت، كما تتيح لنا ذلك توحيد لغة البرمجة ضمن طبقات المشروع كافة ما يزيد التركيز ويوفر إمكانية لاستخدام نفس المكتبات ومشاركة الشيفرة بين الواجهات الأمامية بطرف العميل والواجهة الخلفية على الخادم.

تتميز بيئة نود بطريقة التنفيذ الغير متزامنة asynchronous execution ما يمنحها قوة وأفضلية بالأداء في تنفيذ المهام التي تتطلب غزارة في الدخل والخرج ضمن في تطبيقات الويب أو تطبيقات الزمن الحقيقي، كتطبيقات بث الفيديو أو التطبيقات التي تحتاج لإرسال واستقبال مستمر للبيانات.

سنكتب في هذا الفصل معًا برنامجنا الأول في بيئة تشغيل نود، وسنتعرف على بعض المفاهيم في تلك البيئة التي ستساعدنا في تطوير برنامج يتيح للمستخدم معاينة متغيرات البيئة على النظام لديه، ولتنفيذ ذلك سنتعلم طباعة السلاسل النصية إلى الطرفية console، واستقبال الدخل من المستخدم، ثم الوصول لمتغيرات البيئة environment variables على النظام.

> قد يختلف الإصدار الحالي لديك عن الإصدار الذي استعملناه، ولن تكون هنالك اختلافات أو مشاكل تذكر أثناء تطبيق الأمثلة والشيفرات ولكن إن حصلت إلى خطأ متعلق بتنفيذ شيفرة مطابقة تمامًا لشيفرة شرحناها فتأكد من اختلاف الإصدارات آنذاك وإن كانت المشكلة مرتبطة بها.

1.1 الطباعة إلى الطرفية

المهمة الأولى للمبرمج عند تعلمه للغة برمجة أو تجربة بيئة جديدة هي كتابة برنامج لطباعة عبارة "أهلًا بالعالم!" أو "Hello, World!"، لذا نبدأ بإنشاء ملف جديد نسميه"hello.js" ونفتحه ضمن أي برنامج محرر نصوص تريد كبرنامج المُفكرة Notepad مثلًا، سنستخدم في هذا الفصل المحرر nano من سطر الأوامر كالتالي:

```
nano hello.js
```

نكتب الشيفرة التالية داخله ونحفظ الملف:

```
console.log("Hello World");
```

يوفر الكائن `console` في بيئة نود في السطر السابق توابع تمكننا من الكتابة إلى مجاري الخرج مثل مجرى الخرج القياسي `stdout` أو إلى مجرى الخطأ القياسي `stderr` وغيرهما والتي عادةً تمثل سطر الأوامر، ويطبع التابع `log` القيم المُمررة له إلى المجرى `stdout` لتظهر لنا في الطرفية، حيث أن المجاري في نود هي إما كائنات تستقبل بيانات مثل المجرى `stdout`، أو تُخرج بيانات كمقبس شبكة أو ملف، وأي بيانات تُرسل إلى المجرى `stdout` أو `stderr` ستظهر مباشرةً في الطرفية، ومن أهم مزايا المجاري سهولة إمكانية إعادة توجيهها، كتوجيه خرج تنفيذ برنامج ما إلى ملف أو إلى برنامج آخر، والآن وبعد التأكد من حفظ الملف والخروج من محرر النصوص، حيث إذا كنت تستخدم `nano` اضغط على `CTRL+X` للخروج واضغط `Y` عند سؤالك عن حفظ الملف، وبهذا يكون البرنامج الذي كتبناه جاهزًا للتنفيذ.

1.2 تشغيل البرنامج

نستخدم الأمر `node` لتشغيل البرنامج السابق كالتالي:

```
node hello.js
```

سيتم تنفيذ شيفرات البرنامج داخل ملف hello.js ويظهر الناتج ضمن الطرفية:

```
Hello World
```

ما حدث هو أن مفسر نود قرأ الملف ونفذ التعليمة `console.log("Hello World");` عبر استدعاء التابع `log` من الكائن العام `console`، الذي مررنا له السلسلة النصية `"Hello World"` كوسيط، ونلاحظ عدم طباعة علامات الاقتباس التي مررناها على الشاشة، لأنها ضرورية ضمن الشيفرة فقط لتحديد النص كسلسلة نصية، والآن بعد أن نفذنا برنامجنا البسيط السابق بنجاح، سنطوره ليصبح أكثر تفاعلية.

1.3 استقبال الدخل من المستخدم عبر وسائط سطر الأوامر

يُظهر البرنامج السابق نفس الخرج كل مرة عند تنفيذه، لذا ولجعل الخرج متغيرًا يمكننا جلب المدخلات من المستخدم وعرضها على الشاشة كما هي، وهذا هو مبدأ عمل أدوات سطر الأوامر، حيث أنها تقبل من المستخدم عددًا من الوسائط التي تحدد طريقة عمل البرنامج مثال على ذلك الأمر `node` نفسه، حيث أنه يقبل الوسيط `--version` ليطبع عندها رقم إصدار بيئة نود المثبتة على الجهاز بدلًا من تشغيل مفسر البرامج.

سنقوم بالتعديل على برنامجنا ليستقبل الدخل من المستخدم عن طريق وسائط سطر الأوامر، لهذا نُنشئ ملفًا جديدًا بالاسم arguments.js:

```
nano arguments.js
```

ونكتب داخله الشيفرة التالية ونحفظ الملف:

```
console.log(process.argv);
```

يحوي الكائن العام `process` في نود على توابع وبيانات تتعلق بالإجرائية الحالية، والخاصية `argv` ضمنه هي مصفوفة سلاسل نصية تُمثل عناصرها وسائط سطر الأوامر المُمررة للبرنامج عند تنفيذه، وأصبح بإمكاننا الآن تمرير عدة وسائط إلى البرنامج أثناء تنفيذه كالتالي:

```
node arguments.js hello world
```

لنحصل على الخرج:

```
[
   '/usr/bin/node',
  '/home/hassan/first-program/arguments.js',
  'hello',
  'world'
]
```

يمثل أول وسيط ضمن المصفوفة `process.argv` مسار الملف التنفيذي لنود الذي نفَّذ البرنامج، بينما الوسيط الثاني هو مسار ذلك البرنامج، والوسائط البقية تمثل الوسائط التي أدخلها المستخدم في حالتنا هي كلمة hello وكلمة world، وهي عادةً ما يهمنا عند التعامل مع الوسائط المُمررة للبرنامج وليس الوسائط التي يمررها نود افتراضيًا.

الآن نفتح ملف البرنامج arguments.js مجددًا لنعدل عليه:

```
nano arguments.js
```

ونحذف التعليمة السابقة ونضع بدلًا منها التعليمة التالية:

```
console.log(process.argv.slice(2));
```

بما أن الخاصية argv هي مصفوفة Array، يمكننا الاستفادة من التوابع المتاحة ضمن المصفوفات في جافاسكربت، مثل التابع slice لنختار العناصر التي نريدها فقط من المصفوفة، فنمرر له العدد 2 كوسيط لنحصل على كافة عناصر المصفوفة argv بعد العنصر الثاني والتي تمثل الوسائط التي مررها المستخدم بالضبط.

نعيد تنفيذ البرنامج كما نفذناه آخر مرة ونلاحظ الفرق:

```
node arguments.js hello world
```

سيظهر لنا الخرج التالي:

```
[ 'hello', 'world' ]
```

بعد أن أصبح البرنامج يستقبل الدخل من المستخدم، سنطوره الآن ليعرض لنا متغيرات البيئة المتاحة للبرنامج.

1.4 الوصول لمتغيرات البيئة

سنعرض في هذه الخطوة متغيرات البيئة environment variables المتاحة في النظام وقيمها باستخدام الكائن العام process.env ونطبعها في الطرفية، فمتغيرات البيئة هي بيانات على شكل مفتاح وقيمة key/value مُخزَّنة خارج البرنامج يوفرها نظام التشغيل، حيث يتم تعيين قيمها إما من قبل النظام أو المستخدم، وتكون متاحة لجميع الإجرائيات لاستخدامها كطريقة لضبط إعدادات البرامج أو حالتها أو طريقة عملها، ويمكننا الوصول إليها عن طريق الكائن العام process.

نُنشئ ملفًا جديدًا بالاسم environment.js:

```
nano environment.js
```

ونكتب داخله الشيفرة التالية ونحفظ الملف:

```
console.log(process.env);
```

يحوي الكائن env على متغيرات البيئة المتاحة لحظة تشغيل نود للبرنامج.

ننفذ الآن البرنامج الجديد:

```
node environment.js
```

نلاحظ ظهور خرج مشابه للتالي:

```
{
  SHELL: '/bin/bash',
  SESSION_MANAGER:
'local/hassan-laptop:@/tmp/.ICE-unix/1638,unix/hassan-laptop:/tmp/.ICE
-unix/1638',
  WINDOWID: '0',
  QT_ACCESSIBILITY: '1',
  COLORTERM: 'truecolor',
  XDG_CONFIG_DIRS: '/home/hassan/.config/kdedefaults:/etc/xdg/xdg-
plasma:/etc/xdg:/usr/share/kubuntu-default-settings/kf5-settings',
  GTK_IM_MODULE: 'ibus',
  LANGUAGE: 'en_US:ar',
  SSH_AGENT_PID: '1427',
  PWD: '/home/hassan/first-program',
  LOGNAME: hassan,
  HOME: '/home/hassan',
  IM_CONFIG_PHASE: '1',
  LANG: 'en_US.UTF-8',
  LESSCLOSE: '/usr/bin/lesspipe %s %s',
  TERM: 'xterm-256color',
  USER: 'hassan',
  PATH:
'/home/hassan/.nvm/versions/node/v16.15.1/bin:/usr/local/sbin:/usr/
local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/
snap/bin',
  DBUS_SESSION_BUS_ADDRESS: 'unix:path=/run/user/1000/bus',
  OLDPWD: '/',
  _: '/home/hassan/.nvm/versions/node/v16.15.1/bin/node'
}
```

القيم الظاهرة في متغيرات البيئة في الخرج السابق تعتمد بغالبها على إعدادات نظام التشغيل، لذا ستلاحظ وجود فرق في الخرج عند محاولتك لتنفيذ البرنامج، والآن بدلًا من عرض قائمة بكافة متغيرات البيئة المتاحة سنطور البرنامج للحصول على متغير معين منها فقط.

1.5 الوصول لمتغير بيئة محدد

تُمثل خصائص الكائن `process.env` رابطة بين أسماء متغيرات البيئة وقيمها مُخزَّنة كسلاسل نصية، حيث يمكننا الوصول لأي خاصية ضمن الكائن في جافاسكربت بذكر اسمها بين قوسين مربعين.

نفتح الملف `environment.js` ضمن محرر النصوص ونعدل محتواه:

```
nano environment.js
```

نعدل التعليمة الموجودة فيه لتصبح ما يلي ثم نحفظ الملف:

```
console.log(process.env["HOME"]);
```

ننفذ البرنامج:

```
node environment.js
```

نحصل على خرج كالتالي:

```
/home/hassan
```

بدلًا من طباعة الكائن `process.env` بكل قيمه اخترنا الخاصية `HOME` فقط منه، والتي تمثل مسار مجلد المستخدم الحالي، وهي نفس القيمة التي يمثلها متغير البيئة `$HOME` المتوفر في بيئات يونكس، وستلاحظ اختلافًا في خرج هذا البرنامج أيضًا عند تنفيذه لنفس السبب السابق، حيث سيُعرَض مسار مجلد المستخدم الخاص بك.

والآن بعد أن تعلمنا طريقة الوصول لقيمة متغير بيئة محدد، سنطور البرنامج ليسأل المستخدم عن متغير البيئة الذي يريد عرضه.

1.6 جلب متغير بيئة يحدده المستخدم

سنستفيد من إمكانية وصولنا لوسائط سطر الأوامر التي يُمرِّرها المستخدم، مع إمكانية وصولنا لمتغيرات البيئة لإنشاء أداة سطر أوامر بسيطة مهمتها طباعة قيمة متغير بيئة محدد على الشاشة.

نُنشئ ملفًا جديدًا بالاسم echo.js:

```
nano echo.js
```

ونكتب داخله الشيفرة التالية ونحفظ الملف:

```
const args = process.argv.slice(2);
console.log(process.env[args[0]]);
```

يُخزن السطر الأول من هذا البرنامج جميع الوسائط الـتي مَرَّرهـا المسـتخدم ضـمن ثـابت يـدعى `args`، ثم يطبع السطر الثاني عند تنفيذه متغير بيئة محدد بحسب قيمة أول عنصر من عناصر الثـابت `args`، أي بحسـب أول وسيط مرَّره المستخدم عند تنفيذ البرنامج.

ننفِّذ البرنامج ونمرر له اسم متغير بيئة ما كالتالي:

```
node echo.js HOME
```

سيظهر لنا الخرج التالي:

```
/home/hassan
```

حُفِظ الوسيط `HOME` الذي مرَّرناه للبرنامج السابق ضمن المصفوفة `args`، ثم استخدمناه للعثور على قيمـة متغير البيئة المقابل له باستخدام الكائن `process.env`، وبذلك يصبح بإمكاننا الوصول لقيمـة أي متغـير بيئـة متوفر في النظام، وجرِّب الآن بنفسك وحاول عرض قيم متغيرات البيئة التالية: `PWD` و `USER` و `PATH`.

والآن سنطور البرنامج لعرض عدة متغيرات بيئة معًا يطلبها المستخدم بدلًا من واحد فقط.

1.7 عرض عدة متغيرات بيئة

يمكن للبرنامج الآن في كل مرة عرض متغير بيئة واحد فقط، لذا في هذه الخطوة سنطوره ليستقبل عدد من الوسائط من سطر الأوامر ويعرض متغيرات البيئة المقابلة لها.

نفتح ملف البرنامج `echo.js` ضمن محرر النصوص:

```
nano echo.js
```

ونبدل بمحتواه الشيفرة التالية ثم نحفظ الملف:

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

args.forEach(arg => {
  console.log(process.env[arg]);
});
```

توفر لنا جافاسكربت افتراضيًا التابع forEach ضمن المصفوفات، والذي يقبل تابع رد نداء callback كمعامل له يتم استدعاءه خلال المرور على كل عنصر من عناصر المصفوفة، حيث نلاحظ أننا مررنا للتابع forEach من الكائن args رد نداء يمثل وظيفة طبع قيمة متغير البيئة المقابل للوسيط الحالي.

ننفذ البرنامج السابق ونمرر له عدة أسماء لمتغيرات بيئة كالتالي:

```
node echo.js HOME PWD
```

لنحصل على الخرج:

```
/home/hassan
/home/hassan/first-program
```

نتأكد باستخدامنا للتابع forEach من معالجة كافة الوسائط التي مررها المستخدم للبرنامج، والمُخزنة ضمن الثابت args وطباعة متغير البيئة المقابل لها، وبعد أن أصبح البرنامج الآن يعرض قيم جميع متغيرات البيئة التي يطلبها المستخدم، يجب معالجة الحالة التي يمرر فيها المستخدم متغير بيئة غير موجود.

1.8 معالجة طلب المستخدم لمتغير بيئة غير موجود

لنحاول طلب عرض قيمة متغير بيئة ما غير موجود من البرنامج ونلاحظ ماذا سيحدث:

```
node echo.js HOME PWD NOT_DEFINED
```

نحصل على خرج كالتالي:

```
/home/hassan
/home/hassan/first-program
undefined
```

نلاحظ عرض قيمة أول متغيري بيئة في أول سطرين كما هو متوقع، أما في السطر الأخير ظهرت لنا القيمة undefined، وكما نعلم في جافاسكربت القيمة undefined تعني أن الخاصية أو المتغير غير مُعرَّف ولم تُحدد قيمته بعد، وذلك لأن متغير البيئة الذي طلبناه NOT_DEFINED غير موجود لذا طُبعت تلك القيمة عوضًا، وبدلًا من ذلك يمكننا عرض رسالة خطأ للمستخدم تُعلمه أن متغير البيئة الذي يطلبه غير موجود.

نفتح الملف مرة أخرى للتعديل عليه:

```
nano echo.js
```

ونضيف الشيفرة التالية:

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

args.forEach(arg => {
  let envVar = process.env[arg];
  if (envVar === undefined) {
    console.error(`Could not find "${arg}" in environment`);
  } else {
    console.log(envVar);
  }
});
```

ما قمنا به هو تعديل تابع رد النداء المُمرر للتابع `forEach` ليقوم بالخطوات التالية:

1. استخراج متغير البيئة للوسيط الحالي وتخزين قيمته في المتغير `envVar`.
2. التحقق ما إذا كانت قيمة `envVar` غير مُعرّفة `undefined`.
3. في حال كانت قيمة `envVar` غير مُعرّفة `undefined` نطبع رسالة تُعلم المستخدم بعدم وجود متغير بيئة لهذا الوسيط.
4. في حال عُثر على متغير البيئة نطبع قيمته.

يطبع التابع `console.error` رسالة على الشاشة من خلال مجرى الخطأ القياسي `stderr`، بينما يطبع التابع `console.log` القيم المُمررة له عبر مجرى الخرج القياسي `stdout`، ولن نلاحظ أي فرق بين استخدام المجريين `stdout` و `stderr` عند تنفيذ البرنامج من خلال سطر الأوامر، ويعتبر استخدام كل تابع منهما في حالته الخاصة وتحديدًا طباعة رسائل الخطأ عبر المجرى `stderr` من الممارسات الجيدة في تطوير البرمجيات، لأنه يُمَكّن البرامج الأخرى من تحديد تلك الأخطاء والتعامل معها إن لزم ذلك.

والآن نعيد تنفيذ البرنامج كالتالي:

```
node echo.js HOME PWD NOT_DEFINED
```

لنحصل على الخرج:

```
/home/hassan
/home/hassan/first-program
Could not find "NOT_DEFINED" in environment
```

نلاحظ ظهور رسالة للمستخدم تفيد بأن المتغير NOT_DEFINED لم يُعثر عليه.

1.9 خاتمة

بدأنا في هذا الفصل بكتابة برنامج بسيط لطباعة عبارة بسيطة على الشاشة، وانتهينا بكتابة أداة لسطر الأوامر في نود تعرض للمستخدم متغيرات البيئة التي يطلبها، ويمكنك الآن التطوير على تلك الأداة بنفسك بمحاولة التحقق مثلًا من مدخلات المستخدم قبل طباعة أي قيمة، وإعادة خطأ مباشرةً في حال أن أحد متغيرات البيئة المطلوبة غير موجود، وبذلك سيحصل المستخدم من البرنامج على قيم متغيرات البيئة فقط في حال كانت جميع المتغيرات المطلوبة موجودة.

2. استخدام الوضع التفاعلي REPL

حلقـة اقـرأ-قيّم-اطبـع أو REPL -اختصـارًا للعبـارة Read Evaluate Print Loop- هي صـدفة تفاعليـة interactive shell تعالج تعابير جافاسكربت البرمجية ضمن بيئة نود، حيث تقرأ تلك الصدفة الشـيفرات الـتي يدخلها المستخدم وتُصرّفها ثم تُقيّم نتيجتها وتطبع تلك النتيجة للمستخدم على الشاشة آنيًـا، وتكـرر ذلـك لحين خروج المستخدم من تلك الصدفة، وتأتي REPL مثبتة مسبقًا مع نود، وتسمح لنا باختبار واستكشـاف شـيفرات جافاسكربت داخل بيئة نود بسرعة ودون الحاجة لحفظها أولًا داخل ملف ثم تنفيذها، وسيلزمك في هذا الفصـل للمتابعة معرفة بأساسيات لغة جافاسكربت، ولبيئة نود مُثبتة على الجهاز.

2.1 الدخول والخروج من الوضع REPL

بعد تثبيت نـود على جهـازك، سـيكون وضـع حلقـة REPL متاحًـا للاسـتخدام مباشـرةً، وللـدخول إليـه ننفذ الأمر node فقط ضمن سطر الأوامر كالتالي:

```
node
```

سيدخلنا ذلك في وضع التفاعلي:

```
>
```

حيث يشير الرمز > في بداية السطر لإمكانية إدخالنا شيفرات جافاسكربت لتُعالج، ويمكننا تجربة ذلك بجمع عددين كالتالي:

```
> 2 + 2
```

نضغط زر الإدخال ENTER لتُقيّم صدفة نود ذلك التعبير البرمجي وتطبع نتيجته مباشرةً:

```
4
```

للخـروج من ذلـك الوضـع يمكننـا إمـا كتابـة الأمـر `.exit` أو الضـغط من لوحـة المفـاتيح على الاختصـار CTRL+D، أو الضغط مرتين على الاختصار CTRL+C، للخروج والعودة إلى سطر الأوامر.

والآن بعـد أن علمنـا طريقـة الـدخول والخـروج من الوضـع REPL، سـنتعلم طريقـة تنفيـذ بعض شـيفرات جافاسكربت البسيطة ضمنه.

2.2 تنفيذ شيفرة جافاسكربت ضمن REPL

يمنحنا الوضع REPL التفاعلي طريقة سريعة لاختبار شيفرات جافاسكربت فورًا، ودون الحاجة لإنشـاء ملـف لها أولًا، حيث يمكننا تنفيذ أي تعبير بـرمجي سـليم يمكن تنفيـذه عـادةً ضـمن بيئـة نـود، إذ اختبرنـا في المثـال السابق جمع عددين، ولنختبر الآن تنفيذ قسمة عددين.

ندخل أولًا إلى الوضع REPL كما تعلمنا:

```
node
```

ونُدخل التعبير البرمجي ونضغط زر الإدخال لتنفيذه:

```
> 10 / 5
```

نحصل على الخرج التالي وهو ناتج العملية السابقة:

```
2
```

يمكن أيضًا مثلًا تنفيذ العمليات على السلاسل النصية ولنختبر ذلك بتنفيذ ضم سلسلتين نصيتين كالتالي:

```
> "Hello " + "World"
```

وسيظهر لنا نتيجة ضم السلسلتين:

```
'Hello World'
```

نلاحظ ظهور النص في النتيجة محاطًا بعلامات اقتباس مفردة بدلًا من علامـات الاقتبـاس المزدوجـة، ففي جافاسكربت لا يؤثر نوع علامات الاقتباس على قيمة السلسلة النصـية، لـذا يسـتخدم الوضـع REPL عنـد إظهـار نتيجة فيها سلسلة نصية علامات الاقتباس المفردة دومًا.

2.2.1 استدعاء التوابع

يستخدم التابع العام `console.log` أو توابع طباعة الرسائل المشابهة له كثيرًا في بيئة نود، حيث يمكننا داخل REPL استدعاء التوابع أيضًا، فلنجرب مثلًا أمر طباعة رسالة كالتالي:

```
> console.log("Hi")
```

سيُستدعى التابع وتظهر نتيجة التنفيذ التالية:

```
Hi
undefined
```

يمثل السطر الأول نتيجة استدعاء التابع `console.log`، والذي يطبع الرسالة إلى المجرى `stdout` والذي يمثل الشاشة، ولأن الوظيفة طباعة وليس إعادة عبر التعبير `return` كنتيجة لتنفيذ التابع، نلاحظ عدم وجود علامات الاقتباس حولها، بينما السطر الثاني يعرض القيمة `undefined` وهي النتيجة التي أعادها التابع بعد انتهاء تنفيذه.

2.2.2 تعريف متغيرات

تُستخدم المتغيرات variables أيضًا بكثرة خلال كتابتنا للشيفرات البرمجية ولا نكتفي بالتعامل مع القيم مباشرة، لذا يتيح لنا REPL إمكانية تعريف المتغيرات تمامًا كما لو كنا نكتبها ضمن ملفات جافاسكربت، ويمكننا اختبار ذلك كالتالي:

```
> let age = 30
```

تظهر لنا النتيجة التالية بعد ضغط زر الإدخال:

```
undefined
```

كما لاحظنا سابقًا عند استدعاء التابع `console.log` كانت القيمة التي يعيدها هي `undefined`، وهنا أيضًا جرى تعريف المتغير `age` ولم نُعد أي قيمة، وسيكون ذلك المتغير متاحًا حتى الانتهاء والخروج من جلسة REPL الحالية، ولاختبار ذلك نستخدم المتغير `age` ضمن عملية ما ولتكن ضربه بعدد كالتالي:

```
> age * 2
```

تظهر لنا نتيجة العملية بعد الضغط على زر الإدخال:

```
60
```

نلاحظ أن REPL يعيد ويطبع لنا نتيجة التعبير البرمجي فورًا، لذا لا نحتاج لاستخدام التابع `console.log` في كل مرة نريد طباعة قيمة على الشاشة، حيث سيطبع تلقائيًا أي قيمة يعيدها الأمر المُدخل.

2.2.3 إدخال الشيفرات متعددة الأسطر

يدعم REPL أيضًا إدخال الشيفرات متعددة السطر، ولنختبر ذلك ننشئ تابعًا يضيف القيمة 3 إلى العدد المُمرر له، ونبدأ تعريفه بإدخال أول سطر منه كالتالي:

```
const add3 = (num) => {
```

وبعد الضغط على زر الإدخال ستلاحظ تغير الرمز `>` في أول السطر إلى رمز النُقط الثلاث:

```
...
```

يلاحظ REPL وجود قوس معقوص `{` في نهاية الأمر المدخل، ما يشير إلى وجود بقية له، فيتم إضافة هامش من النقط وانتظار إدخالنا لباقي الأمر، وذلك لتسهيل القراءة حيث يضيف REPL ثلاث نقط ومسافة في السطر التالي، ليبدو أن الشيفرة يسبقها مسافة بادئة، ونكمل إدخال سطر جسم الدالة، ثم سطر قوس الإغلاق لإنهاء تعريف التابع، ونضغط زر الإدخال بعد كل سطر منها:

```
return num + 3;
}
```

وبعد إدخال آخر سطر الحاوي على قوس الإغلاق للتابع، ستظهر لنا القيمة `undefined`، والتي تدل على القيمة المُرجعة من أمر إسناد الدالة إلى الثابت، ونلاحظ عودة الرمز في بداية السطر إلى رمز إدخال الأوامر `>` بدلًا من النقط `...` وتظهر لنا قيمة الأمر المُدخل:

```
undefined
>
```

يمكننا الآن استخدام الدالة التي عرفناها `add3()` بتمرير قيمة لها كالتالي:

```
> add3(10)
```

ويظهر لنا نتيجة الإضافة التي تعيدها الدالة كالتالي:

```
13
```

يمكن الاستفادة من REPL في تجربة شيفرات جافاسكربت واللعب بها قبل إضافتها إلى النظام أو المشروع الذي نعمل عليه، حيث يوفر REPL اختصارات تساعدنا خلال تلك العملية سنتعرف عليها في الفقرة التالية.

2.3 التعرف على الاختصارات في REPL

يوفر REPL عدة اختصارات تسهل عملية ادخال الشيفرات وتـوفر الـوقت، فمثلًا يحفـظ REPL -كمـا معظم الصدفات- سجلًا بالأوامر المدخلة سابقًا لنتمكن من الرجوع إليها بدلًا من إعادة إدخالها يدويًا مرة أخرى.

جرب مثلًا كتابة القيمة النصية الطويلة التالية:

```
> "The answer to life the universe and everything is 32"
```

يظهر لنا النص نفسه كنتيجة لذلك الأمر:

```
'The answer to life the universe and everything is 32'
```

الآن إذا أردنا إدخال النص السابق نفسه، لكن مع اختلاف وهو تبديل العـدد 32 إلى 42، فيمكننـا ذلـك عـبر الضغط على مفتاح السهم العلوي UP من لوحة المفاتيح للوصول إلى آخر قيمة أدخلناها:

```
> "The answer to life the universe and everything is 32"
```

بعدها يمكننا تحريك المؤشر داخل النص وإزالة العدد 3 وتبديله إلى 4 ونضغط زر الإدخال ENTER مجددًا:

```
'The answer to life the universe and everything is 42'
```

يمكن بالضغط المستمر على السهم العلوي UP الرجوع في سجل تـاريخ الأوامـر المدخلـة سـابقًا واحـد تلـو الآخر، وبالمقابل يمكن الضغط على مفتاح السهم السفلي DOWN للتقـدم إلى الأمـام في سـجل تـاريخ الأوامـر، ويمكن بعد الانتهاء من تفحص سجلات الأوامر المُخزنة الضغط مـرارًا على مفتـاح السـهم السـفلي DOWN إلى حين العودة إلى سطر الإدخال الفارغ لكتابة أمر جديد، ويمكن الوصول إلى قيمة آخر نتيجة عـبر محـرف الشـرطة سفلية _، ولاختبار ذلك نكتب الرمز _ ثم نضغط على زر الإدخال:

```
> _
```

سيظهر لنا السلسلة النصية التي أدخلناها مؤخرًا:

```
'The answer to life the universe and everything is 42'
```

يتيح REPL أيضًا ميزة الإكمال التلقائي للتوابع والمتغيرات والكلمات المفتاحية أثنـاء كتابـة الشـيفرة، فمثلًا إذا أردنا استخدام التابع العام `Math.sqrt` لحساب الجذر التربيعي لعدد يمكننا فقط كتابة الأحـرف الأولى لـذلك الاستدعاء كالتالي مثلًا:

```
> Math.sq
```

ثم الضغط على زر الجدولة تاب TAB ليكمل لنا REPL كتابة باقي اسم التابع بشكل صحيح كالتالي:

```
> Math.sqrt
```

وعندما يكون هناك أكثر من طريقة لإكمال الأمر، سيظهر لنا جميع الاحتمالات الممكنة، فمثلًا إذا حاولنا استدعاء تابع ما من الصنف `Math` كالتالي:

```
> Math.
```

بالضغط على زر الجدولة مرتين سيظهر لنا جميع الاحتمالات الممكنة للإكمال التلقائي لذلك الأمر:

```
> Math.
Math.__defineGetter__       Math.__defineSetter__
Math.__lookupGetter__
Math.__lookupSetter__       Math.__proto__          Math.constructor
Math.hasOwnProperty         Math.isPrototypeOf
Math.propertyIsEnumerable
Math.toLocaleString         Math.toString           Math.valueOf

Math.E                      Math.LN10               Math.LN2
Math.LOG10E                 Math.LOG2E              Math.PI
Math.SQRT1_2                Math.SQRT2              Math.abs
Math.acos                   Math.acosh              Math.asin
Math.asinh                  Math.atan               Math.atan2
Math.atanh                  Math.cbrt               Math.ceil
Math.clz32                  Math.cos                Math.cosh
Math.exp                    Math.expm1              Math.floor
Math.fround                 Math.hypot              Math.imul
Math.log                    Math.log10              Math.log1p
Math.log2                   Math.max                Math.min
Math.pow                    Math.random             Math.round
Math.sign                   Math.sin                Math.sinh
Math.sqrt                   Math.tan                Math.tanh
Math.trunc
```

بحيث تظهر النتيجة السابقة بتنسيق يناسب حجم نافذة سطر الأوامر من ناحية عدد الأعمدة والأسطر لتلك الاحتمالات، وتمثل تلك الاحتمالات جميع التوابع أو الخاصيات المتاحة ضمن الوحدة `Math`.

يمكننا في أي وقت الحصول على سطر فارغ جديد لإدخال الأوامر بالضغط على الاختصار CTRL+C، وذلك دون تنفيذ الأمر الجاري كتابته في السطر الحالي.

إن معرفة الاختصارات السابقة يزيد من كفاءة وسرعة كتابة الشيفرات داخل REPL، كما يحتوي أيضًا على أوامر تزيد الإنتاجية سنتعرف عليها في الفقرة التالية.

2.4 أوامر REPL

يوفر REPL كلمات مفتاحية خاصة تساعدنا في التحكم به، ويبدأ كل من تلك الأوامر برمز النقطة . كما سنتعرف عليها.

2.4.1 الأمر help.

لعرض كل الأوامر المتاحة ضمن REPL يمكننا استخدام الأمر `.help` كالتالي:

```
> .help
```

سيظهر لنا جميع الأوامر المتاحة الخاصة بالوضع REPL وهي قليلة لكن مفيدة:

```
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press ^C to abort current expression, ^D to exit the repl
```

يفيد تنفيذ هذا الأمر في حال نسيان الأوامر المتاحة أو وظيفة كل منها.

2.4.2 الأمران break. و clear.

تظهر فائدة الأمران `.break` و `.clear` خلال كتابتنا الشيفرة متعددة الأسطر إذ تساعد على الخروج من ذلك الوضع، ولنختبر ذلك بكتابة أول سطر من حلقة التكرار `for` كالتالي:

```
for (let i = 0; i < 100000000; i++) {
```

بدلًا من إكمال كتابة أسطر ذلك الأمر يمكننا تنفيذ الأمر `.break` أو الأمر `.clear` للخروج:

```
.break
```

سيظهر لنا الرمز > من جديد، ونلاحظ أن REPL استجاب لهذا الأمر وانتقل إلى سطر جديد فارغ دون تنفيذ الشيفرة التي كنا نحاول إدخالها تمامًا كما لو أننا ضغطنا على الاختصار CTRL+C.

2.4.3 الأمران .save و .load

يُمكّننا الأمر `.save` من حفظ كافة الشيفرات التي أدخلناها منذ بداية جلسة REPL الحالية إلى ملف جافاسكربت، بالمقابل يُمكّننا الأمر `.load` من تنفيذ شيفرات جافاسكربت من ملف خارجي داخل REPL، وذلك بدلًا من كتابة تلك الشيفرات يدويًا، و لاختبار ذلك نخرج أولًا من الجلسة الحالية إما بتنفيذ الأمر `.exit` أو باستخدام الاختصار CTRL+D، ونبدأ جلسة REPL جديدة بتنفيذ الأمر `node`، حيث ستحفظ كل الشيفرات التي سنقوم بكتابتها منذ الآن داخل الملف عند استخدامنا لأمر الحفظ `.save` لاحقًا.

نُعرّف مصفوفة من الفواكه:

```
> fruits = ['banana', 'apple', 'mango']
```

في سطر النتيجة سيظهر:

```
[ 'banana', 'apple', 'mango' ]
```

نحفظ الآن المتغير السابق إلى ملف جديد بالاسم `fruits.js` كالتالي:

```
> .save fruits.js
```

ستظهر رسالة تؤكد حفظ الملف بنجاح:

```
Session saved to: fruits.js
```

مكان حفظ ذلك الملف هو نفس مسار المجلد الذي بدأنا منه جلسة REPL من سطر الأوامر، فمثلًا لو كان مسار سطر الأوامر عندها هو مجلد المنزل home للمستخدم، فسيُحفَظ الملف داخل ذلك المجلد.

والآن نخرج من الجلسة الحالية ونبدأ جلسة جديدة بتنفيذ الأمر `node` مرة أخرى، ونُحمِّل ملف `fruits.js` الذي حفظناه سابقًا بتنفيذ الأمر `.load` كالتالي:

```
> .load fruits.js
```

ليظهر لنا:

```
fruits = ['banana', 'apple', 'mango']

[ 'banana', 'apple', 'mango' ]
```

قرأ الأمر `.load` كل سطر داخل ذلك الملف ونفذه تمامًا كطريقة عمل مفسر جافاسكربت، حيث أصبح بإمكاننا الآن استخدام المتغير `fruits` كما لو أننا أدخلناه سابقًا يدويًا ضمن الجلسة الحالية، ولنختبر ذلك ونحاول الوصول لأول عنصر من تلك المصفوفة:

```
> fruits[1]
```

نحصل على الخرج المتوقع:

```
'apple'
```

ويمكن تحميل أي ملف جافاسكربت باستخدام الأمر `.load` مهما كان، وليس فقط الملفات التي نحفظها، لنختبر ذلك بكتابة ملف جافاسكربت بسيط نُنشئ ملفًا جديدًا ونفتحه باستخدام محرر النصوص:

```
nano peanuts.js
```

ثم ندخل ضمنه الشيفرة التالية ونحفظ التغييرات:

```
console.log('I love peanuts!');
```

نبدأ جلسة REPL جديدة من نفس مسار المجلد الحاوي على ملف جافاسكربت `peanuts.js` الجديد بتنفيذ الأمر `node`، ونُحمِّل الملف إلى الجلسة الحالية بتنفيذ التالي:

```
> .load peanuts.js
```

سيُنفذ الأمر `.load` التعبير البرمجي `console` ضمن ذلك الملف ويُظهر الخرج:

```
console.log('I love peanuts!');

I love peanuts!
undefined
>
```

تظهر فائدة كلا الأمرين `.save` و `.load` عند كتابة الكثير من الشيفرات داخل REPL أو عندما نريد حفظ ما أدخلناه خلال الجلسة الحالية ومشاركته ضمن ملف جافاسكربت.

2.5 خاتمة

تتيح لنا بيئة REPL التفاعلية تنفيذ شيفرات جافاسكربت دون الحاجة لإنشاء ملف لها أولًا، كتنفيذ التعابير البرمجية واستدعاء التوابع وتعريف المتغيرات، وتوفر العديد من الاختصارات والأوامر والمزايا الداعمة لتلك العملية، كتنسيق النص تلقائيًا للأوامر متعددة الأسطر، و سجل بتاريخ الأوامر المُدخلة، إلى أوامر المسح أو الحفظ والتحميل، يضيف لك تعلم REPL مهارة قد تحتاج إليها خلال عملك في وقت ما.

3. إدارة الوحدات البرمجية باستخدام npm وملف package.json

الشهرة والاستخدام الواسع لبيئة نود Node.js في تطوير تطبيقات النظم أو الواجهات الخلفية للويب سببها الأساسي مزايا السرعة والأداء العالي للغة جافاسكربت عند التعامل مع الدخل والخرج I/O، واعتمدت عليها العديد من التطبيقات كبيرة الحجم ما زاد تعقيد وصعوبة إدارة اعتمادياتها dependencies، حيث يوفر نود نظام تقسيم الشيفرة والاعتماديات إلى وحدات modules لتنظيمها وحل تلك المشكلة، ومن أبسط أشكالها هي أي ملف جافاسكربت يحوي توابع وكائنات يمكن استخدامها من قبل البرامج أو الوحدات الأخرى.

ويُدعى تجمع عدة وحدات معًا بالحزمة package، وتُدار مجموعة الحزم باستخدام برنامج مخصص لإدارة الحزم من أشهرها مدير حزم نود npm، والذي يأتي افتراضيًا مع نود ويستخدم لإدارة الحزم الخارجية في المشاريع المبنية ضمن نود، ويستخدم أيضًا لتثبيت العديد من أدوات سطر الأوامر ولتشغيل النصوص أو السكربتات البرمجية للمشاريع، فهو يدير تلك الحزم ويخزن معلوماتها ضمن ملف يسمى package.json داخل مجلد المشروع ويحوي على معلومات مثل:

- الحزم التي يعتمد عليها المشروع وأرقام إصداراتها.
- معلومات تصف المشروع نفسه، كاسم المطور ورخصة الاستخدام وغيرها.
- السكربتات البرمجية الممكن تنفيذها، كالتي تؤتمت بعض المهام الخاصة بالمشروع.

تساعد عملية إدارة البيانات الوصفية metadata والاعتماديات الخاصة بمشروع ضمن ملف واحد هو package.json على توحيد تلك المعلومات ومشاركتها خلال مرحلة تطوير أي مشروع برمجي على أي جهاز ومع أي مطور، حيث يُستخدم ذلك الملف من قبل مدير الحزم لإدارة تلك المعلومات تلقائيًا، ونادرًا ما نضطر لتعديل البيانات داخل هذا الملف يدويًا لإدارة الوحدات البرمجية المستخدمة في المشروع.

سنستخدم في هذا الفصل مدير حزم نود npm لإدارة الحزم وسنتعرف بالتفصيل على محتوى ملف package.json ونستخدمه لإدارة الوحدات البرمجية المُثبتة ضمن المشروع، وسنتعلم طريقة عرض الاعتماديات المستخدمة حاليًا وطريقة تحديثها أو إلغاء تثبيتها وفحصها للعثور على المشاكل الأمنية داخلها.

ستحتاج للمتابعة وتطبيق الأمثلة في هذا الفصل لتثبيت بيئة Node.js على جهازك، حيث استخدمنا في هذا الفصل الإصدار رقم 18.3.0 وبذلك يكون قد ثُبت أيضًا مدير الحزم npm.

3.1 إنشاء ملف الحزمة package.json

لنبدأ بإعداد المشروع الذي سنطبق عليه كافة الخطوات اللاحقة، والذي سيكون عبارة عن حزمة لتحديد المواقع سنسميه locator، ووظيفته تحويل عناوين IP إلى اسم البلد المقابل لها، ولن نخوض في تفاصيل تضمين الشيفرة لذلك المشروع بل سيكون تركيزنا على جانب إدارة الحزم والاعتماديات للمشروع فقط، وسنستخدم في ذلك حزمًا خارجية كاعتماديات للمشروع وفي حال أردت تضمين المشروع بنفسك يمكنك استخدامها نفسها.

بدايةً، نُنشئ ملفًا نسميه package.json، سيحوي على البيانات الوصفية للمشروع وتفاصيل الاعتماديات التي سيعتمد عليها، وكما تشير لاحقة ذلك الملف فمحتوياته ستكون مكتوبة بصيغة JSON وهي الصيغة المعتمدة لتخزين البيانات ومشاركتها على شكل كائنات جافاسكربت objects، وتتألف من أزواج من المفاتيح والقيم key/value المقابلة لها.

وبما أن الملف package.json سيحوي العديد من البيانات يمكننا تجنب كتابتها يدويًا ونسخ ولصق قالب جاهز لتلك البيانات من مكان آخر، لهذا فإن أول ميزة سنتعرف عليها في مدير الحزم npm هو الأمر `init` والذي سيسأل عند تنفيذه عدة أسئلة سيبني ملف package.json للمشروع تلقائيًا اعتمادًا على أجوبتنا لها.

3.1.1 استخدام الأمر init

أول خطوة هي إنشاء مجلد للمشروع الذي سنتدرب عليه من سطر الأوامر أو بأي طريقة أخرى ننشئ مجلدًا جديدًا بالاسم locator:

```
mkdir locator
```

وننتقل إليه:

```
cd locator
```

والآن ننفذ أمر تهيئة ملف package.json:

```
npm init
```

> إذا كنا ننوي استخدام مدير الإصدارات Git لإدارة إصدارات المشروع وحفظ ملفاته، ننشئ مستودع Git داخل مجلد المشروع أولًا قبل تنفيذ أمر التهيئة `npm init`، وسيعلم حينها الأمر أن عملية التهيئة لملف الحزمة تتم بداخل مجلد يحوي مستودع Git، وإذا كان عنوان المستودع البعيد متاحًا ضمنه سيتم إضافة قيم للحقول `repository` و `bugs` و `homepage` تلقائيًا إلى ملف package.json، أما في حال تهيئة المستودع بعد تنفيذ أمر التهيئة سنحتاج حينها لإضافة تلك الحقول وتعيين قيمها يدويًا.

بعد تنفيذ الأمر السابق سيظهر الخرج التالي:

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install ` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (locator)
```

أول سؤال سنُسأل عنه هو اسم المشروع `name`، فإن لم تُعط فستأخذ افتراضيًا اسم المجلد للمشروع ونلاحظ دومًا اقتراح القيم الافتراضية بين القوسين `()` وبما أن القيمة الافتراضية هي ما نريدها يمكننا الضغط على زر الإدخال ENTER مباشرةً لقبولها.

السؤال التالي هو عن رقم إصدار المشروع `version`، حيث أنها ضرورية مع اسم المشروع في حال مشاركة الحزمة التي سنطورها في مستودع حزم npm، فتستخدم حزم نود عادة الترقيم الدلالي Semantic Versioning لإصداراتها، وفيها يدل الرقم الأول على الإصدار الأساسي `MAJOR` الذي يشير أنه أجريت تغييرات جذرية على الحزمة، والرقم الثاني يدل على الإصدار الثانوي `MINOR` الذي يشير لإضافة مزايا على الحزمة، والرقم الثالث والأخير يدل على إصدار الترقيع `PATCH` الذي يشير لتصحيح أخطاء ضمن الحزمة.

نضغط على زر الإدخال ENTER لقبول القيمة الافتراضية لأول إصدار من الحزمة وهو `1.0.0`.

الحقل التالي هو حقل الوصف للمشروع `description` وهو شرح مختصر عن المشروع ووظيفته يفيد عند البحث عن تلك الحزمة من قبل المستخدمين إن نُشر على الإنترنت، والحزمة locator التي سنطورها وظيفتها جلب عنوان IP للمستخدم وإعادة اسم البلد الذي ينتمي له هذا العنوان، وهنا يمكننا كتابة وصف معبر عن وظيفة هذه الحزمة باللغة الإنكليزية شبيه بالتالي:

```
Finds the country of origin of the incoming request
```

السؤال التالي هو عن الملف الأساسي أو المدخل للمشروع entry point فعند تثبيت أي حزمة واستخدامها ضمن مشروع آخر واستيرادها فإن أول ما سيُحمّل هو الملف الذي سنحدده في هذا الحقل، وقيمة المسار للملف المحدد في الحقل `main` يجب أن تكون نسبةً لمجلد المشروع الجذري الذي أول ما يحوي فيه الملف package.json، ويمكننا قبول القيمة الافتراضية المقترحة والضغط على زر الإدخال ENTER باعتبار أن الملف index.js سيكون المدخل هنا.

> تستخدم معظم الحزم الملف index.js كمدخل لها، لهذا تعتبر هذه القيمة الافتراضية للحقل `main` كمدخل لوحدات npm، وحتى عند غياب ملف package.json من مجلد الوحدة ستحاول نود افتراضيًا تحميل الملف index.js من مجلد جذر الحزمة المُستخدمة.

السؤال التالي هو عن أمر تنفيذ اختبارات الحزمة test command، وقيمته يمكن أن تكون إما مسار لملف تنفيذي أو أمر لتشغيل اختبارات المشروع، وتستخدم معظم وحدات نود الشهيرة أطر اختبار مثل Mocha أو Jest أو Jasmine أو غيرها لكتابة اختبارات المشروع، ويمكننا ترك قيمة هذا الحقل فارغة بالضغط على زر الإدخال.

سنُسأل بعدها عن عنوان مستودع Git للمشروع، هنا نُدخل مسار المستودع للمشروع الحالي الذي قد يكون مُستضافًا على أحد الخدمات الشهيرة مثل GitHub، ويمكنك ترك قيمته فارغة أيضًا.

سيُطلب منا بعدها إدخال بعض الكلمات المفتاحية كقيمة للحقل `keywords` والقيمة عبارة عن مصفوفة من السلاسل النصية تحوي مصطلحات وكلمات مفتاحية ستفيد المستخدمين عند البحث عن الحزمة عند نشرها عبر الإنترنت، لذا يفضل إدخال بعض الكلمات القصيرة التي تتعلق بعمل الحزمة لتزداد فرصة العثور عليها وظهورها ضمن عمليات البحث، وندخل الكلمات المفتاحية مفصولًا بينها بفاصلة، فمثلًا لمشروعنا يمكن إدخال بعض الكلمات كالتالي `ip,geo,country` ينتج عن ذلك مصفوفة تحوي ثلاث عناصر كقيمة للحقل `keywords` داخل الملف package.json.

الحقل التالي هو اسم صاحب المشروع أو الكاتب والمطور له `author`، حيث يفيد إدخال تلك المعلومة المستخدمين الراغبين بالتواصل معه لأي سبب، مثل اكتشاف ثغرة أو مشكلة في عمل الحزمة، وتكون قيمة هذا الحقل سلسة نصية بالصيغة التالية: `"الاسم <عنوان البريد الإلكتروني> (موقع الويب)"` مثلًا:

```
"Hassan <hassan@example.com> (https://mywebsite.com)"
```

وإدخال عنوان البريد الإلكتروني وموقع الويب اختياريان ويمكن الاكتفاء بإدخال الاسم فقط.

القيمة الأخيرة هي لحقل رخصة الاستخدام `license`، حيث يحدد ذلك الصلاحيات القانونية والحدود المسموح بها استخدام هذه الحزمة أو المشروع، وبما أن أغلب حزم نود مفتوحة المصدر لذا القيمة الافتراضية

المقترحة هي رخصة ISC، لذا يجب قبل تعيين تلك القيمة مراجعة الرخص المتاحة واختيار المناسبة منها للمشروع، ويمكنك الاطلاع على معلومات أكثر على رخص المشاريع المفتوحة المصدر وفي حال كانت الحزمة مطورة للاستخدام الخاص وليست للمشاركة يمكن إدخال القيمة UNLICENSED لتحديد الحزمة كغير مرخصة للاستخدام العام أبدًا، ولمشروعنا الحالي يمكن استخدام القيمة الافتراضية بالضغط على زر الإدخال وإنهاء تهيئة وإنشاء الملف.

سيعرض بعد ذلك الأمر init محتوى ملف package.json الذي سيُنشئه لنراجعه ونتأكد من جميع القيم وسيظهر خرج كالتالي:

```
About to write to /home/hassan/locator/package.json:

{
  "name": "locator",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming
request",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "Hassan <hassan@your_domain> (https://your_domain)",
  "license": "ISC"
}


Is this OK? (yes)</hassan@your_domain>
```

في حال كانت كل البيانات صحيحة نضغط زر الإدخال للإنهاء وإنشاء ملف package.json وبعدها يمكننا تثبيت الوحدات البرمجية الخارجية ليعتمد عليها مشروعنا وتضاف تفاصيلها في ذلك الملف.

3.2 تثبيت الوحدات البرمجية

عند تطوير المشاريع البرمجية عادة ما نفوض المهام التي لا تتعلق بصلب عمل المشروع إلى مكتبات برمجية خارجية متخصصة في ذلك، ما يتيح للمطور التركيز على عمل المشروع الحالي فقط وتطوير التطبيق بسرعة وكفاءة أكبر عبر استخدام الأدوات والشيفرات البرمجية التي طورها الآخرون على مبدأ لا تخترع العجلة من جديد، فمثلًا إذا احتاج مشروعنا locator لإرسال طلب خارجي إلى الواجهة البرمجية API لخدمة تقدم البيانات الجغرافية اللازمة لنا وهنا يمكننا استخدام مكتبة خاصة بإرسال طلبات HTTP مباشرةً بدلًا من كتابة ذلك بأنفسنا، حيث وظيفة المشروع هي تقديم تلك البيانات الجغرافية إلى مستخدم الحزمة فقط.

وأما تفاصيل إرسال طلبات HTTP لا تتعلق بوظيفة الحزمة لذا يمكن تفويضها لمكتبة خارجية جاهزة مختصة بذلك، يمكننا مثلًا استخدام مكتبة axios والتي تساعد في إرسال طلبات HTTP بشكل عملي وسهل، ولتثبيتها ننفذ الأمر التالي:

```
npm install axios --save
```

الجزء الأول من هذا الأمر `npm install` هو أمر تثبيت الحزم، ويمكن اختصارًا تنفيذه كالتالي `npm i`، حيث نمرر له أسماء الحزم التي نرغب بتثبيتها مفصولة بفراغات بينها وفي حالتنا نريد فقط تثبيت حزمة مكتبة `axios`، بعدها واختياريًا يمكن تمرير الخيار `--save` لحفظ المكتبات المُثبتة كاعتماديات للمشروع ضمن ملف package.json وهو السلوك الافتراضي حتى دون ذكر ذلك الخيار، وبعد تثبيت المكتبة سنلاحظ ظهور خرج مشابه للتالي:

```
...
+ axios@0.27.2
added 5 packages from 8 contributors and audited 5 packages in 0.764s
found 0 vulnerabilities
```

والآن باستخدام أي محرر نصوص نعاين محتوى الملف package.json لنلاحظ التغييرات، سنستخدم مثلًا محرر nano كالتالي:

```
nano package.json
```

نلاحظ ظهور خاصية جديدة بالاسم `dependencies` أو الاعتماديات، والتي تحوي على اعتماديات المشروع الحالي:

```
{
  "name": "locator",
  "version": "1.0.0",
```

```
  "description": "Finds the country of origin of the incoming
request",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "Hassan hassan@your_domain (https://your_domain)",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.27.2"
  }
}
```

وإضافة الوحدة البرمجية التي ثبتناها مع رقم إصدارها يحدد للمطورين الآخرين العاملين على نفس المشروع الاعتماديات الخارجية التي يتطلبها تشغيله.

> قد انتبهت إلى وجود الرمز ^ قبل رقم الإصدار لاعتمادية axios، وبما أن الترقيم الدلالي يحوي ثلاثة أرقام وهي الأساسي الجذري MAJOR والثانوي البسيط MINOR والترقيع PATCH فيشير ذلك الرمز إلى تثبيت الإصدار الأساسي للاعتمادية ولا مانع من تغير الإصدار الثانوي البسيط أو إصدار الترقيع أي يمكن تنزيل الإصدار 0.28.0 أو 0.28.1 مثلًا واستخدامه ضمن المشروع، ويمكن استخدام الرمز ~ أيضًا لتثبيت الإصدار الأساسي والثانوي وسماحية تغير إصدار الترقيع فقط أي يُقبَل إصدار 0.27.3 أو 0.27.4 مثلًا.

ويمكننا إغلاق الملف package.json الآن بعد الانتهاء من الاطلاع عليه، وفي حال استخدام محرر nano يمكن الخروج بالضغط على CTRL + X ثم ENTER.

3.2.1 اعتماديات لازمة أثناء تطوير المشروع

اعتماديات التطوير development dependencies هي الاعتماديات التي ستُستخدم فقط خلال مرحلة تطوير المشروع وليس خلال مراحل بناء المشروع ونشره ولا يعتمد عليها خلال مرحلة الإنتاج production وتشبه تلك الدعامات والسلالم والسقالات التي توضع أثناء بناء عمارة ثم تُزال عند الانتهاء، فمثلًا يستخدم المطورون عادة مكتبات لفحص الشيفرات البرمجية وكشف الأخطاء المحتملة وتوحيد تنسيق كتابة الشيفرات أو ما يدعى Linter.

لنجرب تثبيت اعتمادية تطوير لتنقيح صياغة الشيفرات تدعى eslint ضمن المشروع بتنفيذ الأمر التالي:

```
npm i eslint@8.0.0 --save-dev
```

نلاحظ إضافة الخيار `--save-dev` والذي يخبر npm بحفظ الاعتماديات التي نثبتها كاعتمادية تطوير فقط، لاحظ أيضًا إضافة اللاحقة `@8.0.0` بعد اسم الاعتمادية حيث يتم وسم إصدارات المكتبات عند تحديثها، ويدل الرمز @ مدير الحزم npm أن يثبت إصدار معين من تلك الاعتمادية وفي حال تجاهلنا إضافة ذلك الوسم سيتم تثبيت آخر نسخة موسومة متاحة من تلك الاعتمادية، والآن لنعاين ملف package.json مجددًا:

```
nano package.json
```

ونلاحظ تغير محتواه وإضافة اعتمادية التطوير:

```
{
  "name": "locator",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming
request",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "Hassan hassan@your_domain (https://your_domain)",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.19.0"
  },
  "devDependencies": {
    "eslint": "^8.0.0"
  }
}
```

نلاحظ إضافة الاعتمادية `eslint` ضمن الحقل `devDependencies` مع رقم الإصدار الذي حددناه لها.

3.2.2 المجلد node_modules والملف package-lock.json المولدان تلقائيًا

عند أول تثبيت لأي حزمة ضمن مشروع نود سيُنشئ npm تلقائيًا المجلد node_modules ليُخزن ضمنه كل الوحدات البرمجية التي يحتاج إليها المشروع الحالي، وأيضًا سيُنشئ الملف package-lock.json والذي يحوي معلومات عن تفاصيل إصدارات المكتبات المُثبتة في المشروع، ولنتأكد من وجود تلك الملفات ضمن مجلد المشروع يمكننا ذلك بتنفيذ الأمر `ls` في سطر الأوامر لعرض الملفات الموجودة وسيظهر لنا التالي:

```
node_modules    package.json    package-lock.json
```

يحوي المجلد node_modules كافة الاعتماديات المُثبتة في المشروع، وعادة لا نضيف هذا المجلد إلى مستودع المشروع لأن حجم هذا المجلد سيكبر بعد تثبيتنا لعدة اعتماديات، ولأن ملف package-lock.json يحوي داخله أساسًا تفاصيل إصدارات المكتبات المثبتة ضمن مجلد node_modules تمامًا كما هي، ما يجعل وجود ذلك المجلد ضمن مستودع المشروع غير ضروري.

ويحوي الملف package.json على قائمة بالاعتماديات المقبولة لاستخدامها ضمن المشروع، بينما يحوي الملف package-lock.json على كل التغييرات التي تحدث على ملف package.json أو مجلد node_modules ويحوي أيضًا على أرقام إصدارات الحزم المثبتة بدقة، ويمكن إضافة هذا الملف إلى مستودع المشروع عادة بدلًا من مجلد node_modules لأن محتواه يعبر عن جميع اعتماديات المشروع بكافة تفاصيلها.

3.2.3 تثبيت الاعتماديات باستخدام package.json

يمكن باستخدام الملفين package.json و package-lock.json إعداد الاعتماديات المحددة فيهما لبدء أو استئناف العمل على تطوير مشروع مع فريق، ولنفهم ذلك أكثر يمكننا إنشاء مجلد جديد فارغ بجوار مجلد المشروع الحالي بالاسم `cloned_locator` بتنفيذ الأوامر:

```
cd ..
mkdir cloned_locator
```

ثم ننتقل إلى ذلك المجلد:

```
cd cloned_locator
```

ننسخ الآن ملفي package.json و package-lock.json من مجلد المشروع الأصلي locator إلى المجلد الجديد `cloned_locator` بتنفيذ الأمر:

```
cp ../locator/package.json ../locator/package-lock.json .
```

والآن يمكننا تثبيت نفس اعتماديات المشروع الأصلي بتنفيذ الأمر التالي:

```
npm i
```

سيتحقق بعدها npm من وجود ملف package-lock.json داخل المجلد الحالي، وفي حال عدم وجوده سيقرأ محتويات ملف package.json لمعرفة الاعتماديات المطلوب تثبيتها، وعادة تكون عملية التثبيت أسرع عند وجود ملف package-lock.json لأنه يحوي الأرقام الدقيقة لإصدارات الاعتماديات المطلوبة، ولن يحتاج حينها npm للبحث عن أرقام إصدارات تناسب المشروع.

وكما ذكرنا، يمكن تجاهل تثبيت اعتماديات التطوير عند نشر التطبيق في مرحلة الإنتاج، وهي الاعتماديات المذكورة في ملف package.json ضمن الحقل `devDependencies` ولا تؤثر أبدًا على عمل التطبيق، لذا عند تثبيت المشروع خلال عملية نشر التطبيق يمكن تجاهل تثبيت تلك الاعتماديات بتنفيذ أمر التثبيت كالتالي:

```
npm i --production
```

حيث يشير الخيار `--production` إلى تجاهل اعتماديات التطوير خلال عملية تثبيت اعتماديات المشروع، ولن نستعمل هذا الخيار إلا في حالات محدَّدة فقط تتعمل بمرحلة بناء المشروع وتجهيزه للنشر على الإنترنت.

ولا ننسَ أيضًا العودة إلى مجلد المشروع الأساسي قبل لمتابعة تطبيق باقي الأمثلة:

```
cd ../locator
```

3.2.4 تثبيت الحزم على مستوى النظام

ثبتنا حتى الآن الاعتماديات الخاصة بمشروعنا locator، ولكن يمكن استخدام npm أيضًا للتثبيت اعتماديات وحزم على مستوى نظام التشغيل، ما يعني أن الحزمة المثبتة بتلك الطريقة ستكون متاحة للمستخدم في أي مكان ضمن النظام بشكل مشابه للأوامر المتوفرة في سطر الأوامر، حيث تفيد هذه الميزة باستخدام الوحدات البرمجية كأدوات سطر الأوامر لتنفيذ مهام محددة في المشروع، فمثلًا يمكن استخدام مكتبة Hexo من سطر الأوامر من أي مكان بعد تثبيتها لإنشاء موقع لمدونة بمحتوى ثابت، وذلك بتنفيذ أمر التثبيت العام كالتالي:

```
npm i hexo-cli -g
```

كما نلاحظ إذا أردنا تثبيت أي حزمة عامة سنضيف الخيار `-g` -اختصارًا إلى الكلمة Global عام- لنهاية أمر التثبيت فقط.

> قد يظهر خطأ عند محاولة تثبيت حزمة عامة والسبب قد يكون في صلاحيات المستخدم الحالي، لذا قد تحتاج لصلاحيات مستخدم مسؤول وحاول آنذاك فتح الطرفية بصلاحية مسؤول super user أو إذا كنت تستخدم نظام شبيه بيونكس يمكن تنفيذ الأمر كالتالي: `sudo npm i hexo-cli -g`.

ويمكن التأكد من نجاح عملية التثبيت للمكتبة بتنفيذ الأمر التالي:

```
hexo --version
```

سيظهر خرج مشابه للتالي:

```
hexo-cli: 4.3.0
os: linux 5.15.0-35-generic Ubuntu 22.04 LTS 22.04 LTS (Jammy
Jellyfish)
node: 18.3.0
v8: 10.2.154.4-node.8
uv: 1.43.0
zlib: 1.2.11
brotli: 1.0.9
ares: 1.18.1
modules: 108
nghttp2: 1.47.0
napi: 8
llhttp: 6.0.6
openssl: 3.0.3+quic
cldr: 41.0
icu: 71.1
tz: 2022a
unicode: 14.0
ngtcp2: 0.1.0-DEV
nghttp3: 0.1.0-DEV
```

تعلمنا كيف يمكن تثبيت الوحدات البرمجية الخارجية باستخدام npm، وكيف أنه يمكن تثبيت الحزم محليًا إما كاعتمادية إنتاج أو تطوير، وشاهدنا كيف يمكن تثبيت الحزم باستخدام ملف package.json بمفرده أو مع ملف package-lock.json مجهزة مسبقًا لتوحيد تثبيت إصدارات الاعتماديات للمشروع بين أفراد فريق المطورين، وتعلمنا كيف يمكن تثبيت الحزم بشكل عام على النظام باستخدام الخيار `-g` لنتمكن من استخدامها من أي مكان سواء داخل مشروع نود أو خارجه.

والآن بعد ما تعلمناه من طرق لتثبيت الوحدات البرمجية، سنتعلم في الفقرة التالية طرق إدارة تلك الاعتماديات.

3.3 إدارة الوحدات البرمجية

لا يقتصر دور مدير الحزم على تثبيت الوحدات البرمجية بل يتوسع إلى تنفيذ العديد من المهام الأخرى الـتي تتعلق بإدارة الحزم بعد تثبيتها فمثلًا يحوي npm على أكثر من 20 أمـرًا يتعلـق بـذلك، حيث سـنتعرف في هـذه الفقرة على بعضها والتي تقوم بما يلي:

- عرض الوحدات البرمجية المثبتة.
- ترقية الوحدات البرمجية إلى إصداراتها الأحدث.
- إلغاء تثبيت الوحدات البرمجية التي لا نحتاج إليها.
- فحص الوحدات البرمجية لتحديد الثغرات الأمنية وإصلاحها.

سنطبق الأوامر المتعلقة بتلك المهام على مجلد مشروعنا locator، ويمكن تنفيذ نفس تلك المهـام بشـكل عام عبر إضافة الخيار `-g` في نهاية الأوامر، كما فعلنا عند تثبيت حزمة عامة على مستوى النظام سابقًا.

3.3.1 عرض قائمة بالوحدات المثبتة

يمكن معرفة الوحدات البرمجية المُثبتة ضمن مشـروع مـا بتنفيـذ الأمـر `list` أو `ls` الخـاص بمـدير الحـزم npm بدلًا من معاينة الملف package.json يدويًا، وذلك بتنفيذ الأمر كالتالي:

```
npm ls
```

ليظهر لنا خرج مشابه للتالي:

```
├── axios@0.27.2
└── eslint@8.0.0
```

يمكن إضافة الخيار `--depth` لتحديد مستوى عـرض شـجرة الاعتماديـات السـابقة، فمثلًا عنـدما نمـرر لـه القيمة `0` سيظهر لنا الاعتماديات في أول مستوى فقط وهي اعتماديات المشروع الحالي فقط كما لو نفذنا الأمر `npm ls` دون خيارات إضافية، ويمكن إضافة الخيار `--all` لعرض شجرة الاعتماديات كاملة كالتالي:

```
npm ls --all
```

ليظهر خرج مشابه للتالي:

```
├─┬ axios@0.27.2
│ ├── follow-redirects@1.15.1
│ └─┬ form-data@4.0.0
│   ├── asynckit@0.4.0
```

```
│   ├─┬ combined-stream@1.0.8
│   │ └── delayed-stream@1.0.0
│   └─┬ mime-types@2.1.35
│     └── mime-db@1.52.0
└─┬ eslint@8.0.0
  ├─┬ @eslint/eslintrc@1.3.0
  │ ├── ajv@6.12.6 deduped
  │ ├── debug@4.3.4 deduped
  │ ├── espree@9.3.2 deduped
  │ ├── globals@13.15.0 deduped
  │ ├── ignore@5.2.0
  │ ├── import-fresh@3.3.0 deduped
  │ ├── js-yaml@4.1.0 deduped
  │ ├── minimatch@3.1.2 deduped
  │ └── strip-json-comments@3.1.1 deduped

. . .
```

3.3.2 ترقية الوحدات البرمجية

التحديث الدوري للوحدات البرمجية المستخدمة ضمن المشروع مهم جدًا للحصول على آخر الإصلاحات والتحسينات الأمنية عليها، لذلك يمكن استخدام الأمر `outdated` لعرض الوحدات البرمجية التي يتوفر لها تحديثات توافق متطلبات المشروع كالتالي:

```
npm outdated
```

سيظهر خرج كالتالي:

```
Package  Current  Wanted  Latest  Location             Depended by
eslint     8.0.0  8.17.0  8.17.0  node_modules/eslint  locator
```

يحوي العمود الأول `Package` من الجدول السابق على أسماء الحزم الممكن ترقيتها، والعمود الثاني `Current` يُظهر رقم الإصدار الحالي للحزمة المثبتة ضمن المشروع، والعمود `Wanted` يُظهر رقم آخر إصدار يوافق متطلبات المشروع من الحزمة المطلوب ترقيتها والعمود `Latest` يُظهر آخر إصدار منشور من تلك الحزمة وقد لا يوافق متطلبات المشروع، والعمود `Location` يُظهر مسار مجلد الحزمة الحالي، حيث يمكن تمرير الخيار `--depth` أيضًا للأمر `outdated` تمامًا كما فعلنا مع الأمر `ls`، وتكون قيمته الافتراضية هي الصفر.

ونجد من الخرج السابق أن الحزمة `eslint` يمكن ترقيتها إلى إصدار أحدث، لهذا يمكن استخدام أمر الترقية `update` أو اختصاره `up` مع ذكر أسماء الحزم التي نرغب بترقيتها كالتالي:

```
npm up eslint
```

سيُظهر لنا خرج هذا الأمر رقم إصدار النسخة الجديدة المثبتة:

```
removed 7 packages, changed 4 packages, and audited 91 packages in 1s

packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

وللتأكد من ذلك يمكننا الاستفادة من الأمر `npm ls` وتمرير اسم الحزمة `eslint` ليظهر لنا تفاصيل الحزمة المثبتة ضمن المشروع كالتالي:

```
npm ls eslint
```

نلاحظ عند تمرير اسم حزمة معينة للأمر `npm ls` ستظهر لنا شجرة الاعتماديات المثبتة ضمن المشروع لكن ستحوي فقط على ما يخص الحزمة المحددة `eslint`:

```
└─┬ eslint@8.17.0
  └─┬ eslint-utils@3.0.0
    └── eslint@8.17.0 deduped
```

ويمكن ترقية كل الاعتماديات في المشروع باستخدام أمر الترقية دون تحديد اسم أي حزمة كالتالي:

```
npm up
```

3.3.3 إلغاء تثبيت الوحدات البرمجية

يمكن استخدام الأمر `uninstall` الخاص بمدير الحزم npm لإلغاء تثبيت وحدات من المشروع بإزالة الحزمة أو الوحدة تلك من مجلد node_modules ويُحذف اسم تلك الحزمة من قائمة الاعتماديات ضمن الملف package.json وملف package-lock.json.

نضطر في الكثير من الأحيان لإزالة حزم معينة من مشروع نعمل عليه، مثلًا لإزالة حزمة ما بعد تجربتها وتبين أنها لا تحقق المطلوب أو أنها صعبة الاستخدام، فمثلًا لو أن حزمة `axios` التي نستخدمها لم تفي بالغرض المطلوب منها وهو إرسال طلبات HTTP أو أنها صعبة الاستخدام بالنسبة لهذا المشروع يمكن إلغاء تثبيتها بتنفيذ الأمر `uninstall` أو اختصاره `un` وتمرير اسم الحزمة كالتالي:

```
npm un axios
```

نحصل على الخرج:

```
removed 8 packages, and audited 83 packages in 542ms

packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

نلاحظ عدم ظهور اسم الحزمة التي ألغي تثبيتها، لذا نتأكد من ذلك بعرض الحزم المثبتة حاليًا كالتالي:

```
npm ls
```

سنلاحظ من الخرج التالي أن الحزمة `eslint` أصبحت الوحيدة المثبتة ضمن المشروع، ما يـدل على إلغـاء تثبيت حزمة `axios` بنجاح:

```
locator@1.0.0 /home/ubuntu/locator
└── eslint@8.17.0
```

3.3.4 فحص الوحدات وتدقيقها

يُستعمل الأمر `audit` من مدير الحزم npm في تدقيق الحزم وفحصها لعـرض المخـاطر الأمنيـة المحتملـة ضمن شجرة اعتماديات المشروع المثبتة، ولنختبر ذلك مثلًا بتثبيت إصدار قديم من حزمة request كالتالي:

```
npm i request@2.60.0
```

وسنلاحظ فورًا عند تثبيت حزم قديمة منتهية الصلاحية ظهور خرج مشابه للتالي:

```
npm WARN deprecated cryptiles@2.0.5: This version has been deprecated
in accordance with the hapi support policy (hapi.im/support). Please
upgrade to the latest version to get the best features, bug fixes, and
security patches. If you are unable to upgrade at this time, paid
support is available for older versions (hapi.im/commercial).

npm WARN deprecated sntp@1.0.9: This module moved to @hapi/sntp.
Please make sure to switch over as this distribution is no longer
supported and may contain bugs and critical security issues.

npm WARN deprecated boom@2.10.1: This version has been deprecated in
accordance with the hapi support policy (hapi.im/support). Please
upgrade to the latest version to get the best features, bug fixes, and
security patches. If you are unable to upgrade at this time, paid
support is available for older versions (hapi.im/commercial).
```

```
npm WARN deprecated node-uuid@1.4.8: Use uuid module instead
npm WARN deprecated har-validator@1.8.0: this library is no longer
supported
npm WARN deprecated hoek@2.16.3: This version has been deprecated in
accordance with the hapi support policy (hapi.im/support). Please
upgrade to the latest version to get the best features, bug fixes, and
security patches. If you are unable to upgrade at this time, paid
support is available for older versions (hapi.im/commercial).
npm WARN deprecated request@2.60.0: request has been deprecated, see
https://github.com/request/request/issues/3142
npm WARN deprecated hawk@3.1.3: This module moved to @hapi/hawk.
Please make sure to switch over as this distribution is no longer
supported and may contain bugs and critical security issues.

added 56 packages, and audited 139 packages in 4s

packages are looking for funding
  run `npm fund` for details

vulnerabilities (5 moderate, 2 high, 2 critical)

To address all issues, run:
  npm audit fix --force

Run `npm audit` for details.
```

يخبرنا npm بوجود حزم قديمة يُفضل عدم استخدامها ووجود ثغرات ضمن الاعتماديات الحاليـة للمشـروع، ولعرض تفاصيل أكثر عن ذلك يمكننا تنفيذ الأمر:

```
npm audit
```

سيظهر لنا جدولًا يعرض المخاطر الأمنية الموجودة:

```
# npm audit report

bl  <1.2.3
Severity: moderate
Remote Memory Exposure in bl - https://github.com/advisories/GHSA-
pp7h-53gx-mx7r
fix available via `npm audit fix`
```

```
node_modules/bl
  request  2.16.0 - 2.86.0
  Depends on vulnerable versions of bl
  Depends on vulnerable versions of hawk
  Depends on vulnerable versions of qs
  Depends on vulnerable versions of tunnel-agent
  node_modules/request

cryptiles  <=4.1.1
Severity: critical
Insufficient Entropy in cryptiles -
https://github.com/advisories/GHSA-rq8g-5pc5-wrhr
Depends on vulnerable versions of boom
fix available via `npm audit fix`
node_modules/cryptiles
  hawk  <=9.0.0
  Depends on vulnerable versions of boom
  Depends on vulnerable versions of cryptiles
  Depends on vulnerable versions of hoek
  Depends on vulnerable versions of sntp
  node_modules/hawk

. . .

vulnerabilities (5 moderate, 2 high, 2 critical)

To address all issues, run:
  npm audit fix
```

نلاحظ ظهور مسارات لتلك الثغرات واقتراح npm طرقًا لسدها إما بتحديث تلك الاعتماديات أو تنفيذ الأمر الفرعي `fix` للأمر `audit` لإصلاح المشاكل تلقائيًا كما هو مقترح، ولنجرب ذلك الأمر ونرى ما يحصل:

```
npm audit fix
```

يظهر لنا:

```
npm WARN deprecated har-validator@5.1.5: this library is no longer
supported
```

```
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or
higher.  Older versions may use Math.random() in certain
circumstances, which is known to be problematic.  See
https://v8.dev/blog/math-random for details.
npm WARN deprecated request@2.88.2: request has been deprecated, see
https://github.com/request/request/issues/3142


added 19 packages, removed 34 packages, changed 13 packages, and
audited 124 packages in 3s


packages are looking for funding
  run `npm fund` for details


found 0 vulnerabilities
```

نفذ npm ترقية لحزمتين موجودتين ما أدى لحل المشاكل الأمنية الموجودة، مع ذلك لا زال هناك ثلاث حزم ضمن المشروع قديمة ويفضل عدم استخدامها، وبهذا نرى أن الأمر `audit fix` لا يُصلح كافة المشاكل الموجودة دومًا، وذلك لأن حل تلك المشاكل يتطلب ترقية الحزم إلى إصدارات أعلى والتي قد تؤدي بدورها إلى حصول تعارض في شجرة الاعتماديات مما يتسبب بمشاكل توقف عمل المشروع كله، ولكن يمكن إجبار npm على ترقية تلك الحزم بتمرير الخيار `--force` وحل جميع تلك المشاكل كالتالي:

```
npm audit fix --force
```

ولا ينصح بتنفيذ ذلك لما يسببه من مشاكل في التوافقية بين الاعتماديات كما ذكرنا.

3.4 خاتمة

تعلمنا في هذا الفصل طريقة ترتيب نود للوحدات البرمجية ضمن حزم، وكيف يدير مدير حزم نود npm تلك الحزم، وكيف أن المشاريع في نود تستخدم الملف package.json لتعريف اعتماديات المشروع وإدارتها بالإضافة إلى تخزين بيانات تصف المشروع نفسه.

واستخدمنا أمر npm من سطر الأوامر لتثبيت وترقية وإزالة الوحدات البرمجية وعرض شجرة الاعتماديات للمشروع وللتحقق من إمكانية ترقية الوحدات البرمجية القديمة، وهدف كل ذلك إعادة استخدام الوحدات البرمجية بين المشاريع بدلًا من إعادة كتابتها لتسريع عملية تطوير، حيث يمكنك الآن كتابة الوحدات البرمجية الخاصة بك ومشاركتها مع الآخرين لاستخدامها في مشاريعهم الخاصة، ويمكنك التدرب على ما تعلمته في هذا الفصل بالبحث عن بعض الحزم التي تخدم مشكلة ما تحاول حلها وتثبيتها واختبارها، فمثلًا يمكنك تجربة استخدام TypeScript لإضافة مزايا على لغة جافاسكربت، أو تحويل موقع ويب تعمل عليه إلى تطبيق جوال باستخدام Cordova.

4. إنشاء وحدات برمجية Modules

الوحدة البرمجية module في نود Node.js هي أجزاء من شيفرات جافاسكربت منعزلة قابلة للاستخدام في أكثر من تطبيق، حيث يعد الغرض من الوحدة البرمجية هو تقسيم منطقي لوظيفة عمل الشيفرة، فأي ملف أو مجموعة ملفات يمكن اعتبارها وحدة برمجية في حال أمكن استخدام البيانات والتوابع فيها من قبل برامج أخرى خارجية.

ويَنشأ عن التقسيم الوظيفي للشيفرات بتلك الطريقة وحدات برمجية أخرى يمكن إعادة استخدامها في عدة مشاريع أكبر أو مع مطورين آخرين أي أن الوحدات تبنى باعتماد بعضها على بعضها الآخر بطريقة هرمية، ما يؤدي لتطوير برمجيات غير مترابطة سهلة التطوير والتوسع وتوفر درجة من التعقيد أعلى من الوحدات المكونة لها، ما يفتح بابًا للمساهمة بمشاركة تلك الوحدات البرمجية والتي توفر بيانات وتوابع مفيدة مع مجتمع نود، وهي الطريقة التي جرى فيها تحزيم ونشر كل الوحدات البرمجية على مستودع npm، لهذا كمبرمج نود من الضروري أن تتعلم طريقة إنشاء الوحدات البرمجية.

أخذنا في الفصل السابق فكرة أساسية عن ماهية الوحدات في نود وتعرفنا على مدير حزم نود npm وأهمية الملف package.json لإدارة الوحدات التي يعتمد عليها مشروعنا، وسنتعلم في هذا الفصل كيفية إنشاء وحدة برمجية وظيفتها اقتراح الألوان على مطور الويب لاستخدامها في التصميم، فسنخزن الألوان المتاحة في مصفوفة داخل الوحدة وسنوفر تابعًا للمستخدمين يختار لهم إحداها عشوائيًا، بعدها سنتعلم عدة طرق يمكننا بها استيراد تلك الوحدة واستخدامها ضمن تطبيقات ومشاريع نود الأخرى.

يلزمك في هذا الفصل معرفةً باستخدام حلقة REPL التي يوفرها نود، حيث سنستخدمها لاختبار الوحدة التي سنطورها، لذا يفضل الاطلاع على الفصل الثاني إن لم تطلع عليه مسبقًا.

4.1 إنشاء وحدة برمجية في Node.js

سنشرح في هذه الفقرة طريقة إنشاء وحدة برمجية جديدة في نود، حيث ستحتوي الوحدة التي سنطورها على مصفوفة من الألوان وتابع يختار إحداها عشوائيًا ويعيده للمستخدم، وسنستخدم في ذلك خاصية التصدير `exports` في نود لإتاحة التابع والمصفوفة للبرامج الخارجية.

بدايةً، لنعتمد هيكلية معينة للبيانات التي سنخزنها ضمن الوحدة، حيث سنمثل كل لون بكائن سيحوي الخاصية `name` التي تعبر عن اسم ذلك اللون بصيغة مقروءة، والخاصية `code` وهي سلسلة نصية تمثل ترميز ذلك اللون لاستخدامه في HTML، والصيغة المعتمدة لتمثيل الألوان في HTML هي ستة أرقام بالترميز الست عشري.

نبدأ باختيار بعض تلك الألوان التي ستوفرها وحدتنا البرمجية ونضعها في مصفوفة بالاسم `allColors` وليكن عددها ستة ألوان كما ستحتوي وحدتنا على تابع بالاسم `getRandomColor()` لاختيار لون عشوائي من تلك المصفوفة وإعادته.

ننتقل إلى الخطوات العملية، ننشئ مجلدًا جديدًا لاحتواء المشروع نسميه colors وننتقل إليه كالتالي:

```
mkdir colors
cd colors
```

نُهيئ ملف الحزمة package.json ضمن مجلد المشروع لتتمكن باقي البرامج من استيراده واستخدامه لاحقًا كالتالي:

```
npm init -y
```

يمكن باستخدام الخيار `-y` تخطي الأسئلة التي تظهر عادةً عند تخصيص محتوى ملف الحزمة package.json، وفي حال كنا ننوي نشر تلك الوحدة يجب تخصيص القيم داخل ذلك الملف كما شرحنا في الفصل السابق.

سنحصل بعد تنفيذ الأمر على الخرج التالي:

```
{
  "name": "colors",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
```

```
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

الآن نُنشئ ملف جافاسكربت جديد سيحوي على شيفرة الوحدة البرمجية وسيكون المدخل لها، ونفتحه باستخدام أي محرر نصوص أو شيفرات برمجية، مثلًا باستخدام `nano` كالتالي:

```
nano index.js
```

نبدأ بتعريف الصنف `Color` والذي سنمرر له اسم اللون وترميزه الذي سيستخدم ضمن HTML كالتالي:

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}
```

بعد تعريف هيكلية البيانات التي ستمثل اللون، نُنشئ من ذلك الصنف بعض الكائنات ونخزنها ضمن مصفوفة الألوان كالتالي:

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}

const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];
```

بعدها نُعرّف الدالة التي ستجلب لنا لونًا عشوائيًا عند استدعاءها، لتصبح الشيفرة بالكامل كالتالي:

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}

const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];

exports.getRandomColor = () => {
  return allColors[Math.floor(Math.random() * allColors.length)];
}

exports.allColors = allColors;
```

تشير الكلمة المفتاحية `exports` إلى كائن عام توفره نود لكل وحدة برمجية، حيث ستكون كل الكائنات والتوابع المُعرّفة كخصائص ضمن ذلك الكائن متاحة عند استيراد هذه الوحدة واستخدامها من قبل الوحدات البرمجية الأخرى، ولذلك لاحظ كيف عرّفنا التابع `getRandomColor()` مباشرةً كخاصية ضمن الكائن `exports`، وبعدها أضفنا الخاصية `allColors` ضمن ذلك الكائن التي تشير قيمتها إلى مصفوفة الألوان `allColors` المُنشئة سابقًا.

بناءً على ما سبق، ستتمكن أي وحدة برمجية أخرى بعد استيرادها لهذه الوحدة من الوصول إلى التابع `getRandomColor()` والمصفوفة `allColors` واستخدامهما، وبهذا نكون قد أنشأنا وحدة برمجية توفر للوحدات الأخرى مصفوفة من الألوان وتابعًا يختار إحداها عشوائيًا لتتمكن من استخدامهما.

سنستخدم في الفقرة التالية الوحدة التي طورناها ضمن تطبيق آخر لنفهم فائدة الكائن `export` أكثر.

4.2 اختبار الوحدة البرمجية باستخدام REPL

يفضل قبل البدء باستخدام هذه الوحدة اختبارها أولًا للتأكد من صحة عملها، فسنستخدم في هذه الفقرة الوضع التفاعلي REPL لتحميل الوحدة colors واستدعاء التابع `getRandomColor()` التي توفره لنختبر صحة عمله.

نبدأ أولًا جلسة REPL جديدة ضمن مجلد المشروع الحاوي على الملف index.js كالتالي:

```
node
```

نلاحظ ظهور الرمز `>` في بداية السطر عند الدخول إلى وضع REPL ويمكن الآن إدخال أوامر وشيفرات جافاسكربت لتنفيذها فورًا كما يلي:

```
colors = require('./index');
```

سيُحمّل التابع `require()` الوحدة colors وتحديدًا ملف المدخل entry point لها بعد الضغط على زر الإدخال ENTER لتنفيذ السطر السابق ونلاحظ ظهور الخرج التالي:

```
{
  getRandomColor: [Function],
  allColors: [
    Color { name: 'brightred', code: '#E74C3C' },
    Color { name: 'soothingpurple', code: '#9B59B6' },
    Color { name: 'skyblue', code: '#5DADE2' },
    Color { name: 'leafygreen', code: '#48C9B0' },
    Color { name: 'sunkissedyellow', code: '#F4D03F' },
    Color { name: 'groovygray', code: '#D7DBDD' }
  ]
}
```

ظهرت لنا قيمة الوحدة البرمجية colors التي تم استيرادها، وهي عبارة عما صدّرناه منها، حيث يعيد التابع `require` عند استدعائه قيمة الكائن `exports` من الوحدة المستوردة وهي colors في حالتنا، والذي أضفنا إليه داخلها تابعًا بالاسم `getRandomColor()` وخاصيةً بالاسم `allColors`، وهو ما ظهر ضمن الخرج، ويمكننا الآن اختبار التابع `getRandomColor()` كالتالي:

```
colors.getRandomColor();
```

نلاحظ كيف أعاد لنا لونًا عشوائيًا:

```
Color { name: 'groovygray', code: '#D7DBDD' }
```

سيظهر لك لونًا مختلفًا عند تنفيذ الأمر في كل مرة، وذلك لأن الاختيار عشوائي، والآن بعد إتمام الاختبار يمكننا الخروج من جلسة REPL بتنفيذ أمر الخروج التالي الذي سيعيدنا إلى سطر الأوامر:

```
.exit
```

تحققنا في هذه الفقرة من صحة عمل الوحدة البرمجية التي أنشأناها سابقًا وذلك باستخدام REPL، وسنطبق في الفقرة التالية نفس الخطوات لاستيراد واستخدام الوحدة لكن هذه المرة ضمن مشروع حقيقي.

4.3 تثبيت وحدة منشأة محليًا كاعتمادية

استوردنا الوحدة البرمجية أثناء اختبارها ضمن صدفة REPL في الفقرة السابقة بذكر المسار النسبي لها، أي ذكرنا مسار مجلد الملف index.js بدءًا من المسار الحالي، ولا تُعتمد طريقة الاستيراد هذه إلا في حالات خاصة إذ تُستورد الوحدات بذكر أسمائها لتجنب المشاكل التي قد تحدث عند نقل مجلدات المشاريع التي نعمل عليها أو تعديل مساراتها، وسنثبت في هذه الفقرة الوحدة البرمجية colors باستخدام أمر التثبيت `install` من npm، لذلك ننشئ بدايةً وحدة برمجية جديدة خارج مجلد الوحدة colors، بالرجوع إلى المجلد الأب له وإنشاء مجلد جديد كالتالي:

```
cd ..
mkdir really-large-application
```

وننتقل لمجلد المشروع الجديد:

```
cd really-large-application
```

ثم نُهيئ كما تعلمنا سابقًا ملف الحزمة package.json لهذا المشروع بتنفيذ الأمر:

```
npm init -y
```

سيتم توليد ملف package.json بالمحتوى التالي:

```
{
  "name": "really-large-application",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
```

```
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

نثبت الآن الوحدة colors كالتالي:

```
npm install --save ../colors
```

بذلك نكون قد ثبتنا الوحدة colors ضمن المشروع الجديد، ونعاين الآن الملف package.json لنرى كيـف تُحفَظ الاعتماديات المحلية فيه:

```
nano package.json
```

نُلاحظ إضافة سطر جديد ضمن الخاصية dependencies يُذكر فيه اسم الوحدة ومسارها النسبي:

```
{
  "name": "really-large-application",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "colors": "file:../colors"
  }
}
```

حيث نُسخِت الوحدة colors إلى مجلد الاعتماديات node_modules للمشروع الجديد، ويمكننا التأكد من ذلك باستعراض محتوياته باستخدام الأمر التالي:

```
ls node_modules
```

يظهر اسم مجلد الاعتمادية موجودًا ضمنه:

```
colors
```

يمكن الآن استخدام تلك الوحدة ضمن هذا المشروع، لذلك نُنشئ ملف جافاسكربت جديد:

```
nano index.js
```

ونستورد بدايةً الوحدة colors ونستخدم منها الدالة `getRandomColor()` لاختيار لون عشوائي، ثم نطبع رسالة إلى الطرفية تخبر المستخدم باللون الذي يمكنه استخدامه، لذا نكتب داخل الملف index.js الشيفرة التالية:

```
const colors = require('colors');


const chosenColor = colors.getRandomColor();
console.log(`You should use ${chosenColor.name} on your website. It's
HTML code is ${chosenColor.code}`);
```

نحفظ الملف ونخرج منه، والآن عند تنفيذ هذا البرنامج سيخبرنا بلون عشوائي يمكننا استخدامه:

```
node index.js
```

نحصل على خرج مشابه للتالي:

```
You should use leafygreen on your website. It's HTML code is #48C9B0
```

بهذا نكون ثبتنا الوحدة البرمجية colors ضمن المشروع ويمكننا التعامل معها وإدارتها كأي اعتمادية أخرى ضمن المشروع، لكن يجب الانتباه أنه في كل مرة نعدل شيفرة الوحدة colors مثلًا لإضافة ألوان جديدة يجب علينا حينها تنفيذ أمر الترقية `npm update` ضمن مشروع التطبيق لتحديث الاعتمادية واستخدام المزايا الجديدة، ولتجنب تكرار تنفيذ ذلك عند كل تعديل في الفقرة التالية سنستخدم الوحدة colors بطريقة مختلفة تمكننا من استخدام أحدث إصدار لها ضمن المشاريع المعتمدة عليها أثناء العمل عليها وتطويرها.

4.4 ربط وحدة محلية

قد نمر في حالة نعمل فيها على تطوير وحدة برمجية محليًا ونستخدمها في الوقت نفسه ضمن مشروع آخر، وسيصعب آنذاك ترقيتها باستمرار ضمن المشروع كما أشرنا سابقًا، والحل يكمن في ربط الوحدات البرمجية بدلًا من تثبيتها لاستخدامها مباشرة وهي قيد التطوير والبناء.

سنتعلم ذلك في هذه الفقرة عن طريق ربط الوحدة colors ضمن التطبيق الذي يستخدمها، وسنختبر الربط بإجراء تعديلات على الوحدة colors ونتحقق من التحديث الآني لتلك التعديلات ضمن اعتمادية التطبيق دون الحاجة للترقية أو لتثبيت الوحدة من جديد، لذلك نزيل بدايةً تثبيت الوحدة من التطبيق بتنفيذ الأمر التالي:

```
npm un colors
```

يربط مدير الحزم npm الوحدات البرمجية مع بعضها باستخدام الوصلات الرمزية symbolic links والتي تمثل مؤشرًا يشير إلى ملف أو مجلد ما ضمن نظام الملفات، ويُنفذ الربط هذا على مرحلتين:

1. إنشاء وصلة أو رابط عام global link للوحدة حيث يُنشئ npm وصلة رمزية بين مجلد الوحدة البرمجية ومجلد الاعتماديات العام node_modules الذي تُثبَّت فيه كل الحزم العامة على مستوى النظام كله، أي الحزم المُثبَّتة باستخدام الخيار `-g`.
2. إنشاء وصلة محلية local link بحيث يُنشئ npm وصلة رمزية بين المشروع المحلي وبين الرابط العام للوحدة البرمجية المراد استخدامها فيه.

ننشئ الرابط العام بالدخول إلى مجلد الوحدة colors واستخدام الأمر `link` كالتالي:

```
cd ../colors
sudo npm link
```

سيظهر لنا خرج كالتالي:

```
/usr/local/lib/node_modules/colors -> /home/hassan/colors
```

أنشِئت بذلك وصلة رمزية في مجلد node_modules العام تشير إلى مجلد الوحدة colors، والآن نعود إلى مجلد المشروع really-large-application لربط الوحدة ضمنه كالتالي:

```
cd ../really-large-application
sudo npm link colors
```

سنلاحظ ظهور خرج مشابه للتالي:

```
/home/hassan/really-large-application/node_modules/colors ->
/usr/local/lib/node_modules/colors -> /home/hassan/colors
```

> يمكن اختصار الأمر `link` بكتابة `ln` بدلًا منه، ليصبح أمر الربط كالتالي `npm ln colors` وسنحصل على نفس النتيجة.

وكما يُظهر خرج أمر الربط السابق فقد أنشِئت وصلة رمزية في مجلد node_modules للمشروع really-large-application تشير إلى الوصلة الرمزية لمجلد الوحدة colors الموجودة في مجلد node_modules العام على مستوى النظام، والتي بدورها تشير إلى مجلد الوحدة colors الفعلي، وبهذا تكون عملية الربط اكتملت ويمكن تشغيل ملف المشروع للتأكد بأن الربط صحيح ولا زال المشروع يعمل كما هو:

```
node index.js
```

نحصل على خرج مشابه للتالي:

```
OutputYou should use sunkissedyellow on your website. It's HTML code
is #F4D03F
```

نلاحظ عدم تأثر المشروع ولا زال يعمل كما هو، والآن لنختبر ما إذا كانت التعديلات على الوحدة التي طورناها ستنعكس مباشرة ضمن المشروع الذي يستخدمها، لذلك نفتح الملف index.js الخاص بالوحدة colors ضمن محرر النصوص:

```
cd ../colors
nano index.js
```

ونضيف مثلًا دالةً جديدةً مهمتها جلب درجة من درجات اللون الأزرق من الألوان المتوفرة، ولا تحتاج لتمرير معاملات وستعيد العنصر الثالث من مصفوفة الألوان المحلية `allColors` مباشرةً والذي هو من درجات اللون الأزرق، لذا نضيف الأسطر الأخيرة إلى الملف كالتالي:

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}

const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];

exports.getRandomColor = () => {
        return allColors[Math.floor(Math.random() *
allColors.length)];
        }
```

```
exports.allColors = allColors;

exports.getBlue = () => {
  return allColors[2];
}
```

نحفظ الملف ونخرج منه، ونفتح ملف index.js ضمن مجلد المشروع really-large-application:

```
cd ../really-large-application
nano index.js
```

ونستخدم داخله الدالة الجديدة `getBlue()` المضافة إلى الوحدة ونطبع إلى الطرفية جملة تحوي خصـائص ذلك اللون كالتالي:

```
const colors = require('colors');

const chosenColor = colors.getRandomColor();
console.log(`You should use ${chosenColor.name} on your website. It's
HTML code is ${chosenColor.code}`);

const favoriteColor = colors.getBlue();
console.log(`My favorite color is ${favoriteColor.name}/$
{favoriteColor.code}, btw`);
```

نحفظ الملف ونخرج منه، وبذلك يصبح المشروع يستخدم التابع الجديد الـذي أنشـأناه `getBlue()`، والآن ننفذ البرنامج ونرى النتيجة:

```
node index.js
```

سنحصل على خرج مشابه لما يلي:

```
OutputYou should use brightred on your website. It's HTML code is
#E74C3C
My favorite color is skyblue/#5DADE2, btw
```

نلاحظ كيف تمكنا من استخدام آخر التعـديلات الـتي أجريناهـا ضـمن الوحـدة colors مباشـرةً دون الحاجـة لتنفيذ أمر الترقية `npm update` لتلك الوحدة، حيث يسهل ذلك عملية تطوير الوحدات البرمجيـة ويخفـف من تكرار تنفيذ نفس الأوامر بكثرة.

حاول التفكير دومًا عند تطوير التطبيقات الكبيرة والمعقدة نسبيًا كيف يمكن تجميع الشيفرات التي يتم تطويرها ضمن وحدات برمجية منفصلة تعتمد على بعضها، ويمكن إعادة استخدامها في عدة مشاريع، أما في حال كانت الوحدة البرمجية تستخدم فقط ضمن برنامج واحد عندها يفضل إبقاءها ضمن نفس مجلد المشروع ذاك وربطها عن طريق المسار النسبي لها.

وأما في حال التخطيط لمشاركة الوحدة بشكل منفصل لاحقًا أو في استخدامها في مشروع مختلف عن المشروع الحالي فانظر إن كان الربط أنسب لحالتك أم التثبيت كما تعلمت إلى الآن، إذ الفائدة الأكبر من ربط الوحدات قيد التطوير استخدام أحدث إصدار منها دومًا دون الحالة لترقيتها كل حين، وإلا فمن الأسهل تثبيتها باستخدام الأمر `npm install`.

4.5 خاتمة

غصنا عميقًا في هذا الفصل في وحدات نود والتي هي مجموعة من التوابع والكائنات في جافاسكربت خصوصًا في كيفية استخدامها من قبل البرامج الأخرى، فأنشأنا وحدة برمجية وحددنا داخلها بعض الدوال والكائنات كخصائص للكائن `exports` لإتاحتها للاستخدام من قبل التطبيقات الخارجية، واستوردنا تلك الوحدة إلى برنامج جديد واستخدمناها ضمنه.

أصبح بإمكانك الآن استخراج بعض المكونات من البرامج التي تعمل عليها إلى وحدات برمجية منفصلة بتحديد ما تود إعادة استخدامها ضمنها، وبذلك تجمع البيانات والتوابع الخاصة بها معًا ضمن وحدة منفصلة وتعزلها عن باقي التطبيقات مما يمكنك من إعادة استخدامها وتطويرها وحتى مشاركتها مع الآخرين، وكلما كتبت وحدات أكثر وصقلت مهارتك البرمجية فيها، اكتسبت خبرة كبيرة تخولك من تطوير برامج نود عالية الجودة.

5. طرق كتابة شيفرات غير متزامنة التنفيذ

البرمجة المتزامنة synchronous programming في جافاسكربت تعني تنفيذ التعليمات في الأسطر البرمجية سطرًا تلو الآخر بحسب ترتيب كتابتها تمامًا، ولكن لا حاجة للالتزام بترتيب التنفيذ هذا دومًا، فمثلًا عند إرسال طلب عبر الشبكة ستضطر الإجرائية التي يُنفذ فيها البرنامج إلى انتظار رد ذلك الطلب ووصول جوابه قبل أن نتمكن من إكمال تنفيذ باقي البرنامج، حيث وقت انتظار إتمام الطلب هذا هو وقت مهدور، هنا يأتي دور البرمجة اللامتزامنة asynchronous programming لتحل هذه المشكلة، حيث تُنفذ فيها الأسطر البرمجية للبرنامج بترتيب مختلف عن ترتيب كتابتها الأصلي، فيصبح بإمكاننا مثلًا في مثالنا السابق تنفيذ تعليمات برمجية أخرى في أثناء انتظار إتمام عملية إرسال الطلب ووصول جوابه المنتظر مع البيانات المطلوبة.

تُنفَذ شيفرة جافاسكربت ضمن خيط وحيد thread ضمن الإجرائية، حيث تعالج شيفراتها بشكل متزامن ضمن ذلك الخيط عبر تنفيذ تعليمة واحدة فقط في كل لحظة، ويتوضح أثر البرمجة المتزامنة في هذه الحالة أكثر، فعند تنفيذ المهام التي تحتاج لوقت كبير ضمن ذلك الخيط سيُعيق ذلك تنفيذ كل الشيفرات اللاحقة لحين انتهاء تلك المهمة، لذا وبالاستفادة من مزايا برمجة جافاسكربت اللامتزامنة يمكننا إزاحة المهام التي تأخذ وقتًا طويلًا في التنفيذ إلى خيط آخر في الخلفية وبالتالي حل المشكلة، وبعد انتهاء تلك المهمة الطويلة تُنفذ الشيفرات المتعلقة بمعالجة بياناتها ضمن الخيط الأساسي لشيفرة جافاسكربت مجددًا.

سنتعلم في هذا الفصل طرق إدارة المهام اللامتزامنة باستخدام حلقة الأحداث Event Loop الخاصة بجافاسكربت والتي تُنهي بواسطتها مهامًا جديدة أثناء انتظار انتهاء المهام الأخرى، ولذلك سنطور برنامجًا يستفيد من البرمجة اللامتزامنة لطلب قائمة من الأفلام من الواجهة البرمجية لاستديو Ghibli وحفظ بياناتها ضمن ملف CSV، حيث سننفذ ذلك بثلاثة طرق وهي دوال رد النداء callback functions والوعود promises وأخيرًا باستخدام اللاتزامن والانتظار async/await ومع أنه من غير الشائع حاليًا استخدام دوال رد النداء في البرمجة اللامتزامنة في جافاسكربت، إلا أنه من المهم تعلم تلك الطريقة لفهم تاريخ الانتقال لاستخدام الوعود

ووجودها أساسًا، ثم تأتي آلية اللاتزامن والانتظار لتسمح باستخدام الوعود بطريقة أبسط، وهي الطريقة المعتمدة حاليًا عند كتابة الشيفرات اللامتزامنة في جافاسكربت.

5.1 حلقة الأحداث Event Loop

لنتعرف بدايةً على الطريقة التي ينفذ بها جافاسكربت الدوال داخليًا، ما سيسمح لنا لاحقًا بفهم أكثر عند كتابة الشيفرات اللامتزامنة وتزيد قدرتنا على استكشاف الأخطاء وتصحيحها حين حدوثها، حيث يضيف مفسر جافاسكربت كل دالة تُنفَّذ إلى مكدس الاستدعاءات call stack، وهو هيكلية بيانات شبيهة بالقائمة بحيث يمكن إضافة أو حذف العناصر منه من الأعلى فقط أي تعتمد مبدأ الداخل آخرًا يخرج أولًا LIFO -اختصارًا إلى Last in, first out- فعند إضافة عنصرين إلى المكدس مثلًا يمكن حذف آخر عنصر تمت إضافته أولًا، فمثلًا عند استدعاء الدالة `functionA()` سيُضاف ذلك إلى مكدس الاستدعاء، وإذا استدعت الدالة `functionA()` داخلها دالة أخرى مثلًا `functionB()` فسيضاف الاستدعاء الأخير لأعلى مكدس الاستدعاء، وبعد الانتهاء من تنفيذه سيُزال من أعلى مكدس الاستدعاء، أي ينفذ جافاسكربت أولًا الدالة `functionB()` ثم يزيلها من المكدس عند انتهائها، ثم يُنهي تنفيذ الدالة الأب `functionA()` ثم يزيلها أيضًا من مكدس الاستدعاء، لهذا يتم دومًا تنفيذ الدوال الأبناء أو الداخلية قبل الدوال الآباء أو الخارجية.

عندما يُنفذ جافاسكربت عملية لا متزامنة ككتابة البيانات إلى ملف مثلًا، فسيضيفها إلى جدول خاص ضمن الذاكرة يُخزَّن فيه العملية وشرط اكتمالها والدالة التي ستُستدعى عند اكتمالها، وبعد اكتمال العملية ستضاف تلك الدالة إلى رتل الرسائل message queue، وهو هيكلية بيانات تشبه القائمة أيضًا تُضاف إليها العناصر من الأسفل وتزال من الأعلى فقط أي تعتمد مبدأ الداخل أولًا يخرج أولًا FIFO -اختصارًا إلى First in, First out- وحين انتهاء عمليتين لا متزامنتين والتجهيز لاستدعاء الدوال الخاصة بهما سيتم استدعاء الدالة الخاصة بالعملية التي انتهت أولًا، حيث تنتظر الدوال ضمن رتل الرسائل إضافتها إلى مكدس الاستدعاء.

وتبقى حلقة الأحداث في فحص دائم لمكدس الاستدعاء بانتظار فراغه، عندها يُنقل أول عنصر من رتل الرسائل إلى مكدس الاستدعاء، ويعطي جافاسكربت الأولوية للدوال ضمن رتل الرسائل بدلًا من استدعاءات الدوال الجديدة التي يفسرها ضمن الشيفرة، وبذلك تسمح تركيبة عمل مكدس الاستدعاء ورتل الرسائل وحلقة الأحداث بتنفيذ شيفرات جافاسكربت أثناء معالجة المهام اللامتزامنة.

والآن بعد أن ألقينا نظرة عامة على حلقة الأحداث وتعرفنا فيها على طريقة تنفيذ الشيفرات اللامتزامنة في جافاسكربت يمكننا البدء بكتابة شيفرات لا متزامنة باستخدام إحدى الطرق لذلك، إما بدوال رد النداء أو الوعود أو باستخدام اللاتزامن والانتظار async/await.

5.2 البرمجة اللامتزامنة باستخدام دوال رد النداء

تُمرَّر دالة رد النداء callback function كمعامل للدوال الأخرى وتُنفَذ عند انتهاء تنفيذ الدالة المُمررة لها، وتحوي دالة رد النداء عادة شيفرات لمعالجة نتيجة تلك العملية أو شيفرات لتنفيذها بعد انتهاء تنفيذ العملية اللامتزامنة، حيث استخدمت هذه الطريقة لفترة طويلة وكانت أشيع طريقة مستخدمة لكتابة الشيفرات اللامتزامنة، ولكنها لم تعد مستخدمة حاليًا لأنها تُصعّب قراءة ووضوح الشيفرة، لكن سنستخدمها في هذه الفقرة لكتابة شيفرة جافاسكربت لا متزامنة لنتعرف بذلك على كل الطرق الممكنة ونلاحظ الفروقات بينها وميزة كل منها، حيث نستخدم دوال رد النداء بأكثر من طريقة ضمن الدوال الأخرى، وعادة ما نُمرِّرها كآخر معامل للدالة اللامتزامنة كالتالي:

```
function asynchronousFunction([ Function Arguments ], [ Callback
Function ]) {
    [ Action ]
}
```

لكننا لسنا ملزومين باتباع هذه البنية عند كتابة الدوال اللامتزامنة، ولكن شاع تمرير دالة رد النداء كآخر معامل للدالة اللامتزامنة ليسهل التعرف عليه بين المبرمجين، وتُمرَّر عادة دالة رد النداء كدالة مجهول الاسم -التي تُعرّف بلا اسم- إذ يُحسِّن تمريرها كآخر معامل قراءة الشيفرة، ولنفهم ذلك أكثر سننشئ وحدة برمجية في نود وظيفتها كتابة قائمة من أفلام استوديو Ghibli إلى ملف، فنبدأ بإنشاء مجلد سيحوي ملف جافاسكربت للبرنامج وملف الخرج النهائي كالتالي:

```
mkdir ghibliMovies
```

ندخل إلى المجلد:

```
cd ghibliMovies
```

سنرسل بدايةً طلب HTTP للواجهة البرمجية لاستديو Ghibli ونطبع داخل دالة رد النداء نتيجة ذلك الطلب، ولنتمكن من ذلك نحتاج إلى مكتبة تساعدنا في إرسال طلبات HTTP والوصول إلى البيانات ضمن الرد باستخدام دالة رد نداء، لذا نُهيئ ملف الحزمة للوحدة بتنفيذ الأمر التالي:

```
npm init -y
```

ونثبت مكتبة request بتنفيذ الأمر:

```
npm i request --save
```

ننشئ ملفًا جديدًا بالاسم callbackMovies.js ونفتحه باستخدام أي محرر نصوص:

```
nano callbackMovies.js
```

ونكتب داخله الشيفرة التالية والتي سترسل طلب HTTP باستخدام مكتبة request السابقة:

```
const request = require('request');

request('https://ghibliapi.herokuapp.com/films');
```

> انتبه إلى أن هيروكو قد أوقفت الخدمات المجانية لذا لن يعمل الرابط في الأعلى، يمكنك نسخ الواجهة Studio Ghibli API من مستودعها وتشغيلها محليًا لديك ثم استعمال رابط الواجهة المحلية أو كان صعبًا عليك، فابحث عن واجهة برمجية أخرى.

نُحمّل في أول سطر مكتبة request التي ثبتناها، حيث ستعيد المكتبة دالة يمكن استدعاؤها لإنشاء طلبات HTTP نخزنها ضمن الثابت `request`، ثم نرسل طلب HTTP باستدعاء الدالة `request()` وتمرير عنوان الواجهة البرمجية API له.

لنطبع الآن البيانات من نتيجة الطلب إلى الطرفية بإضافة الأسطر كالتالي:

```
const request = require('request');

request('https://ghibliapi.herokuapp.com/films', (error, response,
body) => {
    if (error) {
        console.error(`Could not send request to API: $
{error.message}`);
        return;
    }
    if (response.statusCode != 200) {
        console.error(`Expected status code 200 but received $
{response.statusCode}.`);
        return;
    }
    console.log('Processing our list of movies');
    movies = JSON.parse(body);
    movies.forEach(movie => {
        console.log(`${movie['title']}, ${movie['release_date']}`);
    });
});
```

مررنا للدالة `request()` معاملان هما عنوان URL للواجهة البرمجية API لإرسال الطلب إليها، ودالة رد نداء سهمية لمعالجة أي أخطاء قد تحدث أو معالجة نتيجة إرسال الطلب عند نجاحه بعد انتهاء تنفيذه.

لاحظ أن دالة رد النداء أخذت ثلاثة معاملات وهي كائن الخطأ `error` و كائن الرد `response` وبيانات جسم الطلب `body`، فبعد اكتمال الطلب سيُعيَّن قيم لتلك المعاملات بناءً على النتيجة، ففي حال فشل الطلب سيتم تعيين قيمة كائن للمعامل `error`، وتعيين القيمة `null` لكل من `response` و `body`، وعند نجاح الطلب سيتم تعيين قيمة الرد للمعامل `response`، وفي حال احتوى الرد على بيانات في جسم الطلب ستُعيَّن كقيمة للمعامل `body`.

ونستفيد من تلك المعاملات داخل دالة رد النداء التي مَرَّرناها لها للتحقق من وجود الخطأ أولًا، ويفضل التحقق من ذلك دومًا ضمن دوال رد النداء بحيث لا نكمل تنفيذ باقي التعليمات عند حدوث خطأ، وفي حال وجود خطأ سنطبع رسالة الخطأ إلى الطرفية وننهي تنفيذ الدالة، بعدها نتحقق من رمز الحالة للرد المرسل.

في حال عدم توافر الخادم للرد على الطلب أو تغيير الواجهة البرمجية أو إرسال طلبية خاطئة سنلاحظ ذلك من رمز الرد ويمكن التحقق من نجاح العملية وسلامة الرد بالتحقق من أن رمز الحالة يساوي `200` ويمكننا بذلك متابعة معالجة الطلب، وفي حالتنا عالجنا الطلب بتحليل الرد وتحويله إلى مصفوفة ثم طبع كل عنصر من عناصرها -التي تمثل الأفلام- على شكل اسم الفلم وتاريخ إصداره، والآن نحفظ الملف وتخرج منه وننفذه كالتالي:

```
node callbackMovies.js
```

ليظهر الخرج كالتالي:

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
```

```
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

حصلنا على قائمة بأفلام من إنتاج استوديو Ghibli مع تواريخ إصدارها بنجاح، والآن نريد من البرنـامج كتابـة القائمة إلى ملف، لذا نعدل الملف callbackMovies.js ضمن محرر النصوص ونضيف الأسـطر التاليـة لإنشـاء ملف بصيغة CSV يحوي بيانات الأفلام المجلوبة:

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

request('https://ghibliapi.herokuapp.com/films', (error, response,
body) => {
    if (error) {
        console.error(`Could not send request to API: $
{error.message}`);
        return;
    }

    if (response.statusCode != 200) {
        console.error(`Expected status code 200 but received $
{response.statusCode}.`);
        return;
    }

    console.log('Processing our list of movies');
    movies = JSON.parse(body);
    let movieList = '';
    movies.forEach(movie => {
        movieList += `${movie['title']}, ${movie['release_date']}\n`;
    });

    fs.writeFile('callbackMovies.csv', movieList, (error) => {
        if (error) {
            console.error(`Could not save the Ghibli movies to a file:
${error}`);
```

```
            return;
        }

        console.log('Saved our list of movies to
callbackMovies.csv');;
    });
});
```

نلاحظ بدايةً استيراد الوحدة البرمجية `fs` والتي توفرها بيئة نود للتعامل مع الملفات حيث نريد التابع `writeFile()` منها لكتابة البيانات إلى ملف بطريقة لا متزامنة، وبدلًا من طباعة البيانات إلى الطرفية، يمكننا إضافتها إلى السلسلة النصية للمتغير `movieList` ثم نستدعي التابع `writeFile()` لحفظ قيمة `movieList` النهائية إلى ملف جديد بالاسم callbackMovies.csv، ثم نمرر أخيرًا دالة رد نداء للتابع `writeFile()` حيث سيُمرَّر لها معاملًا وحيدًا وهو كائن الخطأ `error` نعرف بالتحقق منه إذا ما فشلت عملية الكتابة إلى الملف، وذلك مثلًا عندما لا يملك المستخدم الحالي الذي يُنفِّذ إجرائية نود صلاحيات كافية لإنشاء ملف جديد ضمن المسار الحالي.

والآن نحفظ الملف وننفذه مرة أخرى:

```
node callbackMovies.js
```

سنلاحظ ظهور ملف جديد ضمن مجلد المشروع ghibliMovies بالاسم callbackMovies.csv يحوي قائمة أفلام تشبه القائمة التالية:

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
```

```
Ponyo, 2008
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

نلاحظ أننا كتبنا ذلك المحتوى إلى ملف CSV ضمن دالة رد النداء لطلب HTTP المرسل، حيث أن الشيفرات ضمن تلك الدالة ستُنفَذ بعد انتهاء عملية إرسال الطلب فقط، وفي حال أردنا الاتصال بقاعدة بيانات بعد كتابة محتوى ملف CSV السابق يجب إنشاء دالة لا متزامنة أخرى تُستدعَى ضمن دالة رد نداء التابع `writeFile()`، وكلما أردنا تنفيذ عمليات لا متزامنة متلاحقة يجب تغليف المزيد من دوال رد النداء داخل بعضها البعض، فإذا أردنا مثلًا تنفيذ خمس عمليات لا متزامنة متتالية بحيث تُنفَّذ كل منها بعد انتهاء العملية التي تسبقها وسنحصل في النهاية على شيفرة بنيتها تشبه التالي:

```
doSomething1(() => {
    doSomething2(() => {
        doSomething3(() => {
            doSomething4(() => {
                doSomething5(() => {
                    // العملية النهائية
                });
            });
        });
    });
});
```

وبذلك ستصبح داول رد النداء المتداخلة هذه معقدة وصعبة القراءة خصوصًا إن احتوت على أسطر تعليمات عديدة، ويتوضح ذلك خصوصًا في المشاريع الكبيرة والمعقدة نسبيًا، فيصبح من الصعب التعامل مع تلك العمليات وهي نقطة ضعف هذه الطريقة والسبب في عدم استخدامها لمعالجة العمليات اللامتزامنة في وقتنا الحالي، وهنا جاءت الوعود لتحل محلها وتوفر صيغة أفضل في كتابة الشيفرات اللامتزامنة وهذا ما سنتعرف عليه في الفقرة التالية.

5.3 استخدام الوعود لاختصار الشيفرات اللامتزامنة

الوعد Promise هو كائن توفره جافاسكربت وظيفته إرجاع قيمة ما مستقبلًا ومن هنا جاءت تسمية الوعد من أنه يعدك بإعادة قيمة ما لاحقًا، ويمكن للدوال اللامتزامنة أن تُعيد كائن وعد من هذا النوع بدلًا من إرجاع

القيمة النهائية لتنفيذها، وعند تحقق هذا الوعد مستقبلًا fulfilled سنحصل على القيمة النهائية للعملية وإلا سنحصل على خطأ ويكون الوعد قد رُفض rejected، أما خلال التنفيذ يكون الوعد في حالة الانتظار ويتم معالجته.

تستخدم الوعود بالصيغة التالية:

```
promiseFunction()
    .then([ رد نداء يُنفَّذ عند تحقق الوعد ])
    .catch([ رد نداء يُنفَّذ عند رفض الوعد ])
```

نلاحظ أن الوعود تستخدم دوال رد النداء هي أيضًا، حيث نُمرِّر للتابع `then()` دالة رد نداء تُستدعى عند نجاح التنفيذ، ونُمرِّر للتابع `catch()` دالة رد نداء أخرى تُستدعى لمعالجة الأخطاء عند حدوثها أثناء عملية تنفيذ ذلك الوعد.

ولنتعرف على الوعود أكثر سنطور برنامجنا السابق لاستخدام طريقة الوعود بدلًا من دوال رد النداء، ونبدأ بتثبيت مكتبة Axios التي تعتمد على الوعود في عملياتها لإرسال طلبات HTTP:

```
npm i axios --save
```

نُنشئ ملفًا جديدًا بالاسم promiseMovies.js سيحوي النسخة الجديدة من البرنامج:

```
nano promiseMovies.js
```

سنرسل طلب HTTP باستخدام مكتبة axios هذه المرة، وباستخدام نسخة خاصة من وحدة `fs` تعتمد في عملها على الوعود سنحفظ النتيجة ضمن ملف CSV كما فعلنا سابقًا، ونبدأ بكتابة الشيفرة التالية ضمن الملف لتحميل مكتبة Axios وإرسال طلب HTTP للواجهة البرمجية للحصول على قائمة الأفلام:

```
const axios = require('axios');

axios.get('https://ghibliapi.herokuapp.com/films');
```

حملنا في أول سطر مكتبة axios وحفظنا الناتج ضمن الثابت `axios` وبعدها استدعينا التابع `axios.get()` لإرسال طلب HTTP إلى الواجهة البرمجية، حيث سيعيد التابع `axios.get()` وعدًا يمكننا ربطه مع دالة لطباعة الأفلام إلى الطرفية عند نجاح الطلب كالتالي:

```
const axios = require('axios');



axios.get('https://ghibliapi.herokuapp.com/films')
```

```
    .then((response) => {
        console.log('Successfully retrieved our list of movies');
        response.data.forEach(movie => {
            console.log(`${movie['title']}, $
{movie['release_date']}`);
        });
    })
```

بعد إرسال طلب HTTP من نوع GET باستخدام التابع `axios.get()` استخدمنا التابع `then()` والذي سيُنفذ عند نجاح الطلب فقط، وطبعنا داخله الأفلام إلى الطرفية كما فعلنا في الفقرة السابقة، والآن نطور البرنامج لكتابة تلك البيانات إلى ملف جديد باستخدام واجهة للتعامل مع نظام الملفات قائمة على الوعود كالتالي:

```
const axios = require('axios');
const fs = require('fs').promises;


axios.get('https://ghibliapi.herokuapp.com/films')
    .then((response) => {
        console.log('Successfully retrieved our list of movies');
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\
n`;
        });

        return fs.writeFile('promiseMovies.csv', movieList);
    })
    .then(() => {
        console.log('Saved our list of movies to promiseMovies.csv');
    })
```

استوردنا الوحدة البرمجية `fs` مجددًا لكن نلاحظ استخدام الخاصية `.promises` منها، وهي النسخة الخاصة من وحدة `fs` التي تستخدم الوعود كنتيجة لتنفيذ دوالها بدلًا من طريقة دوال رد النداء، وسبب إتاحتها كنسخة منفصلة هو دعم المشاريع التي لازالت تستخدم الطريقة القديمة.

ونلاحظ كيف أصبح أول استدعاء للتابع `then()` يعالج رد الطلب HTTP الوارد ثم يستدعي التابع `fs.writeFile()` بدلًا من طباعة البيانات إلى الطرفية، وبما أننا نستخدم نسخة الوعود من `fs` فسيعيد

التابع `writeFile()` عند استدعائه وعدًا آخر، يجري معالجته باستدعاء `then()` مرة أخرى والتي بـدورها تأخـذ دالة رد النداء تُنفَّذ عند نجاح تنفيذ ذلك الوعد -المُعاد من التابع `writeFile()`.

نلاحظ أيضًا مما سبق أنه يمكن إعادة وعد من داخل وعد آخر، ما سيسـمح بتنفيـذ تلـك الوعـود الواحـد تلـو الآخر، ويوفر لنا ذلك طريقة لتنفيذ عدد من العمليات اللامتزامنة خلف بعضها البعض، وندعو هذه العملية باسـم سلسلة الوعود promise chaining وهي بديل عن استخدام دوال رد النـداء المتداخلـة الـتي تعرفنـا عليهـا في الفقرة السابقة، بحيث يُستدعى التابع `then()` الموالي عند تحقق الوعد المعـاد من سـابقه وهكـذا وعنـد رفض أحد الوعود يُستدعى التابع `catch()` مباشرةً آنذاك وتتوقع السلسلة عن العمل.

> لم نتحقق في هذا المثال من رمز الرد لطلب HTTP الوارد كما فعلنا سابقًا، حيث لن يُلبي axios تلقائيًا الوعد الذي يعيده في حال كان رمز الرد الوارد يمثل أي خطأ، ولذلك لم نعد مضطرين للتحقق منه بأنفسنا.

والآن نضيف التابع `catch()` في نهاية البرنامج لإكماله كالتالي:

```
const axios = require('axios');
const fs = require('fs').promises;


axios.get('https://ghibliapi.herokuapp.com/films')
    .then((response) => {
        console.log('Successfully retrieved our list of movies');
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\
n`;
        });

        return fs.writeFile('promiseMovies.csv', movieList);
    })
    .then(() => {
        console.log('Saved our list of movies to promiseMovies.csv');
    })
    .catch((error) => {
        console.error(`Could not save the Ghibli movies to a file: $
{error}`);
    });
```

في حال فشل أي وعد من سلسلة الوعود تلك سيُنفذ التابع `catch()` تلقائيًا كما أشرنا متجاوزًا أي دوال تسبقه، لذا يمكننا إضافة استدعاء التابع `catch()` مرة واحدة فقط في النهاية لمعالجة أي خطأ قد يحدث من أي عملية سابقة حتى لو كنا ننفذ عدة عمليات غير متزامنة متتالية.

والآن لنتحقق من صحة عمل البرنامج بتنفيذه كالتالي:

```
node promiseMovies.js
```

نلاحظ ظهور نفس البيانات السابقة ضمن الملف promiseMovies.csv:

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

نلاحظ كيف اختصر استخدام الوعود كتابة الكثير من الشيفرات، وكيف أن عملية سَلسَلة الوعود أسهل وأبسط ومقروءة أكثر من طريقة دوال رد النداء المتداخلة، ولكن حتى مع تلك المزايا الجديدة هنالك صعوبات تحصل في حال أردنا تنفيذ العديد من العمليات اللامتزامنة أي ستزداد صعوبة الشيفرة المكتوبة بازدياد طول سلسلة الوعود.

وتحتاج كلا الطريقتين السابقتين سواء دوال رد النداء أو الوعود لإنشاء دوال رد نداء تُعالج ناتج العملية اللامتزامنة، والطريقة الأفضل من ذلك هي انتظار نتيجة العملية اللامتزامنة وتخزينها ضمن متغير خارج أي دالة،

وبذلك يمكننا استخدام النتائج ضمن المتغيرات مباشرةً ودون الحاجة لإنشاء الكثير من الدوال في كل مرة، وهذا تحديدًا ما يميز عملية اللاتزامن والانتظار باستخدام `async` و `await` في جافاسكربت وهي ما سنتعرف عليه في الفقرة التالية.

> إن أردت تعلم المزيد حول الوعود، فارجع إلى توثيق واجهة الوعود Promise

5.4 التعامل مع الوعود باستخدام طريقة اللاتزامن والانتظار async/await

تتيح الكلمة المفتاحية `async` اللاتزامن والكلمة المفتاحية `await` الانتظار صيغة بديلة أبسط للتعامل مع الوعود، إذ ستُعاد النتيجة مباشرةً كقيمة بدلًا من تمريرها على شكل وعد إلى التابع `then()` لمعالجتها وكأننا نستدعي تابع متزامن عادي في جافاسكربت.

ولنخبر جافاسكربت أن دالة ما هي دالة لا متزامنة تُعيد وعدًا، نعرفها بوضع الكلمة المفتاحية `async` قبلها، وبعدها يمكننا استخدام الكلمة المفتاحية `await` داخلها لإخبار جافاسكربت بإرجاع ناتج الوعد المُعاد عند نجاحه بدلًا من إرجاع الوعد نفسه كقيمة، أي تكون صيغة استخدام `async/await` كالتالي:

```
async function() {
    await [عملية غير متزامنة]
}
```

لنطبق استخدامها على برنامجنا ونلاحظ الفرق، لننشئ ملفًا للبرنامج الجديد بالاسم asyncAwaitMovies.js:

```
nano asyncAwaitMovies.js
```

نستورد داخل ذلك الملف نفس الوحدات التي استخدمناها سابقًا لأن طريقة `async/await` تعتمد على الوعود في عملها:

```
const axios = require('axios');
const fs = require('fs').promises;
```

والآن نعرّف دالة باستخدام الكلمة المفتاحية `async` للدلالة على أنها دالة لا متزامنة كالتالي:

```
const axios = require('axios');
const fs = require('fs').promises;
```

```
async function saveMovies() {}
```

عرفنا الدالة `saveMovies()` باستخدام الكلمة المفتاحية `async`، بهذا نستطيع استخدام الكلمة المفتاحية `await` داخلها، أي ضمن الدوال اللامتزامنة التي نعرفها بنفس تلك الطريقة، والآن نستخدم الكلمة المفتاحية `await` لإرسال طلب HTTP إلى الواجهة البرمجية لجلب قائمة الأفلام:

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    let response = await
axios.get('https://ghibliapi.herokuapp.com/films');
    let movieList = '';
    response.data.forEach(movie => {
        movieList += `${movie['title']}, ${movie['release_date']}\n`;
    });
}
```

نرسل طلب HTTP باستخدام `axios.get()` من داخل الدالة `saveMovies()` كما فعلنا سابقًا، لكن لاحظ أنه بدلًا من استدعاء التابع `then()` أضفنا الكلمة المفتاحية `await` قبل الاستدعاء، سينفذ حينها جافاسكربت الشيفرة في الأسطر اللاحقة فقط عند نجاح تنفيذ التابع `axios.get()`، وستُعيَّن القيمة التي يعيدها إلى المتغير `response`، والآن نضيف الشيفرة المسؤولة عن كتابة البيانات الواردة إلى ملف CSV:

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    let response = await
axios.get('https://ghibliapi.herokuapp.com/films');
    let movieList = '';
    response.data.forEach(movie => {
        movieList += `${movie['title']}, ${movie['release_date']}\n`;
    });
    await fs.writeFile('asyncAwaitMovies.csv', movieList);
}
```

نلاحظ استخدامنا للكلمة المفتاحية `await` عند استدعاء التابع `fs.writeFile()` أيضًا لكتابة محتويات الملف، والآن ننهي كتابة الدالة بالتقاط ومعالجة أي أخطاء قد ترميها تلك العمليات باستخدام `try/catch` كما نفعل عادةً في جافاسكربت لالتقاط الأخطاء المرمية:

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    try {
        let response = await
axios.get('https://ghibliapi.herokuapp.com/films');
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\
n`;
        });
        await fs.writeFile('asyncAwaitMovies.csv', movieList);
    } catch (error) {
        console.error(`Could not save the Ghibli movies to a file: $
{error}`);
    }
}
```

وبذلك نضمن معالجة الأخطاء عند حدوثها في العمليات اللامتزامنة داخل جسم `try`، بما فيها أي أخطاء قد تحدث عند إرسال طلب HTTP أو عند فشل الكتابة إلى الملف.

والآن نستدعي الدالة `saveMovies()` اللامتزامنة لضمان تنفيذها عند تنفيذ البرنامج باستخدام نود:

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    try {
        let response = await
axios.get('https://ghibliapi.herokuapp.com/films');
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\
n`;
```

```
        });
        await fs.writeFile('asyncAwaitMovies.csv', movieList);
    } catch (error) {
        console.error(`Could not save the Ghibli movies to a file: ${error}`);
    }
}

saveMovies();
```

لا يوجد فروقات كبيرة بين هذه الطريقة وبين الصيغة العادية لكتابة واستدعاء شيفرات جافاسكربت المتزامنة، حيث لم نحتاج لتعريف العديد من الدوال -تحديدًا دوال ردود النداء- وتمريرها كما فعلنا سابقًا، وتتوضح بذلك ميزة استخدام `async/await` تسهيل قراءة واستخدام الشيفرات اللامتزامنة أفضل من الطرق الأخرى.

والآن ننفذ هذا البرنامج ونختبر عمله:

```
node asyncAwaitMovies.js
```

نلاحظ ظهور ملف جديد بالاسم asyncAwaitMovies.csv ضمن مجلد المشروع ghibliMovies يحوي داخله على التالي:

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
Arrietty, 2010
```

```
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

وبذلك نكون تعرفنا على طريقة عملها استخدام ميزة `async/await` في جافاسكربت.

5.5 خاتمة

تعرفنا في هذا الفصل على الطريقة التي تعالج بها جافاسكربت الدوال وتدير العمليات اللامتزامنة باستخدام حلقة الأحداث، وكتبنا برنامجًا لإنشاء ملف بصيغة CSV بالاعتماد على بيانات واردة من واجهة برمجية API بإرسال طلب HTTP نطلب فيه بيانات عدد من الأفلام مستخدمين بذلك كل طرق البرمجة اللامتزامنة المتوفرة في جافاسكربت، حيث بدأنا ذلك باستخدام طريقة دوال رد النداء القديمة وبعدها تعرفنا على الوعود وطريقة استخدامها، ثم طورنا ذلك باستخدام طريقة اللاتزامن والانتظار `async/await` لتصبح الشيفرة أبسط وأوضح.

ويمكنك الآن بعد ما تعلمته ضمن هذا الفصل استخدام التقنيات التي تعلمتها لكتابة البرامج التي تستخدم العمليات اللامتزامنة، ويمكنك الاستفادة من قائمة الواجهات البرمجية العامة المتاحة لتطوير ما قد يفيدك، وذلك بإرسال طلبات HTTP لا متزامنة إليها كما فعلنا في هذا الفصل.

6. اختبار الوحدات البرمجية باستخدام Mocha و Assert

الاختبارات البرمجية Tests جزء مهم جدًا من عملية تطوير البرمجيات، وهي عبارة عن شيفرات برمجية مهمتها اختبار أجزاء التطبيق خلال مرحلة تطويره للتحقق من أدائها السليم خصوصًا بعد إضافة التطويرات والتعديلات عليه، ولتوفير الوقت عادة ما نؤتمت هذه الاختبارات إذ تمكننا سهولة هذه العملية من تنفيذ تلك الاختبارات باستمرار بعد كل إضافة لشيفرات جديدة على لتطبيق للتأكد من صحة تلك التغييرات وأن أي إضافة أو تعديل على جزء من الشيفرة لا تعطل عمل أي مزايا أخرى موجودة سابقًا، ما يمنح مطور التطبيق الثقة الكافية باعتماد التغييرات خصوصًا قبل مرحلة نشر التطبيق وإتاحته للاستخدام.

ويأتي إطار عمل الاختبارات test framework لينظم طريقة إنشاء وتشغيل حالات الاختبار، ومن أشهر أطر عمل الاختبار تلك في جافاسكربت هو موكا Mocha، حيث تقتصر مهمته على إنشاء وتنظيم الاختبارات وليس تطبيق اختبارات التوكيد assertion testing على عمل الشيفرات، لذا لمطابقة القيم وتطبيق العديد من التوكيدات ضمن الاختبارات نستخدم وحدة برمجية أخرى يوفرها نود لنا افتراضيًا وهي assert.

سنكتب في هذا الفصل اختبارات لتطبيق قائمة مهام سنطوره ضمن بيئة نود، وسنُعد إطار عمل الاختبارات موكا Mocha له ونستخدمه لتنظيم الاختبارات، ثم سنستخدم الوحدة `assert` لكتابة تلك الاختبارات، أي سنستخدم Mocha لتخطيط الاختبارات والوحدة `assert` لتنفيذ التوكيدات ضمن الاختبار، وسيلزمك لتطبيق الأمثلة في هذا الفصل تثبيت بيئة Node.js على جهازك، حيث استخدمنا في هذا الفصل الإصدار رقم 10.16.0، وأيضًا معرفة بأساسيات لغة جافاسكربت.

6.1 كتابة الوحدة البرمجية في نود

نبدأ بكتابة وحدة برمجية سنختبرها لاحقًا وظيفتها إدارة قائمة من المهام وتوفر طريقة لاستعراض قائمة المهام التي نعمل عليها وإضافة مهام جديدة وتحديد المهام المكتملة منها، وستتيح أيضًا ميزة تصدير قائمة

المهام هذه إلى ملف بصيغة CSV، وللتعرف أكثر على طرق كتابة وحدة برمجيـة باسـتخدام نـود يمكنـك مراجعـة الفصل الرابع من هذا الكتاب.

والآن نبدأ بتحضير بيئة العمل وننشئ مجلد باسم المشروع todos:

```
mkdir todos
```

ثم ندخل إلى المجلد:

```
cd todos
```

ونهيّئ ملف حزمة npm لاستخدامه لاحقًا لتنفيذ أوامر الاختبار:

```
npm init -y
```

سنحتاج لاعتمادية واحدة فقط وهي إطار عمل الاختبارات موكا Mocha لتنظيم وتشـغيل الاختبـارات الـتي سنكتبها لاحقًا، لذا ننفذ أمر تثبيتها ضمن المشروع كالتالي:

```
npm i request --save-dev mocha
```

نلاحظ أننا ثبتناها كاعتمادية تطوير لأننا لن نحتاج إليها في مرحلة الإنتاج بل ستستخدم خلال مرحلة التطـوير فقط، ويمكنك التعرف أكثر على طرق إدارة الوحـدات البرمجيـة في Node.js بمراجعـة الفصـل الثـالث من هـذا الكتاب، والآن ننشئ الملف الأساسي لهذه الوحدة كالتالي:

```
touch index.js
```

ونفتحه ضمن أي محرر النصوص وليكن باستخدام محرر نانو nano:

```
nano index.js
```

نبدأ بتعريف الصـنف Todos والـذي سـيحتوي على توابـع سنسـتخدمها لإدارة قائمـة المهـام، لـذا نضـيف الأسطر التالية إلى ملف index.js:

```
class Todos {
    constructor() {
        this.todos = [];
    }
}

module.exports = Todos;
```

عرّفنا في بداية الملف الصنف `Todos` والتابع الباني له `constructor()` بدون أي معاملات، حيث يمكننا إنشاء كائن جديد من هذا الصنف دون الحاجة لتمرير أي قيم له، ومهمته حاليًا إنشاء الخاصية `todos` وتعيين مصفوفة فارغة كقيمة لها، ثم صدّرنا هذا الصنف باستخدام الكائن `modules` في النهاية، كي تتمكن باقي الوحدات البرمجية من استيراد واستخدام الصنف `Todos`، فبدون ذلك لا يمكن لملف الاختبار الذي سنُنشئه لاحقًا استيراد واستخدام هذا الصنف، والآن نضيف تابعًا وظيفته إرجاع مصفوفة المهام المخزنة ضمن الكائن كالتالي:

```
class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }
}

module.exports = Todos;
```

يعيد التابع `list()` نسخة من المصفوفة المخزنة ضمن الصنف باستخدام صيغة التفكيك في جافاسكربت لأن إعادة المتغير `this.todos` مباشرة يعني إعادة مؤشر إلى المصفوفة الأصلية ضمن الصنف `Todos` وبذلك نمنع الوصول إلى المصفوفة الأصلية وإجراء تعديلات عليها عن طريق الخطأ.

> المصفوفات في جافاسكربت تُمرَّر بالمرجعية reference (وكذلك الكائنات objects أيضًا)، أي عند إسناد مصفوفة إلى متغير فإنه يحمل إشارة إلى تلك المصفوفة الأصلية وليس المصفوفة نفسها أي عند استعمال هذا المتغير لاحقًا أو تمريره كمعامل لتابع ما، فستشير جافاسكربت إلى المصفوفة الأصلية دومًا وستنعكس التعديلات عليها، فمثلًا إذا عند إنشاء مصفوفة تحوي ثلاث عناصر أسندناها إلى متغير x، ثم أنشأنا المتغير y وأسندنا له قيمة المصفوفة السابقة كالتالي `y = x`، فسيشير عندها كل من y و x إلى نفس المصفوفة وكل تغيير نقوم به على المصفوفة عن طريق المتغير y سيؤثر على المصفوفة التي يشير إليها المتغير x والعكس صحيح أي كلاهما يشيران إلى المصفوفة نفسها.

والآن لنضيف التابع `add()` ووظيفته إضافة مهمة جديدة إلى قائمة المهام الحالية:

```
class Todos {
    constructor() {
        this.todos = [];
    }
```

```
    list() {
        return [...this.todos];
    }

    add(title) {
        let todo = {
            title: title,
            completed: false,
        }

        this.todos.push(todo);
    }
}

module.exports = Todos;
```

يأخذ التابع `add()` معاملًا من نوع سلسلة نصية ويضعها ضمن خاصية العنوان `title` لكائن المهمة الجديدة، ويعين خاصية اكتمال هذه المهمة `completed` بالقيمة `false` افتراضيًا، ثم يضيف ذلك الكائن إلى مصفوفة المهام الحالية ضمن الكائن.

ومن المهام الأخرى التي يجب أن يوفرها صنف مدير المهام هو تعيين مهمة كمهمة مكتملة، حيث سننفذ ذلك بالمرور على عناصر مصفوفة المهام `todos` والبحث عن عنصر المهمة التي يريد المستخدم تعيينها كمهمة مكتملة، وعند العثور عليها نعينها كمكتملة وإذا لم يُعثر عليها نرمي خطأ كإجراء احترازي، والآن نضيف هذا التابع الجديد `complete()` كالتالي:

```
class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }

    add(title) {
        let todo = {
```

```
            title: title,
            completed: false,
        }

        this.todos.push(todo);
    }

    complete(title) {
        let todoFound = false;
        this.todos.forEach((todo) => {
            if (todo.title === title) {
                todo.completed = true;
                todoFound = true;
                return;
            }
        });

        if (!todoFound) {
            throw new Error(`No TODO was found with the title: "${title}"`);
        }
    }
}

module.exports = Todos;
```

نحفظ الملف ونخرج من محرر النصوص، ونكون بذلك قد انتهينا من كتابة صنف مدير مهام بسيط سنستخدمه لاحقًا لتنفيذ الاختبارات عليه، حيث سنبدأ بالاختبارات اليدوية أولًا في الفقرة التالية لنتأكد من صحة عمله.

6.2 اختبار الشيفرة يدويًا

في هذه الفقرة سننفذ شيفرات التوابع السابقة لصنف إدارة المهام `Todos` يدويًا لنعاين ونتفحص خرجها ونتأكد من عملها كما هو متوقع منها أن تعمل، وتدعى هذه الطريقة بالاختبار اليدوي manual testing فهي أشيع طريقة يطبقها المطورون معظم الوقت حتى لو لكن يكن ذلك مقصودًا، وسنؤتمت لاحقًا تلك العملية باستخدام موكا Mocha لكن بدايةً سنختبر الشيفرات يدويًا لنتعرف على هذه الطريقة ونلاحظ ميزة استخدام إطار خاص لأتمتة الاختبارات.

نبدأ بإضافة مهمتين جديدتين ونعيّن إحداهما كمكتملة، لذلك نبدأ جلسة REPL جديدة ضمن مجلد المشروع نفسه الحاوي على الملف `index.js` كالتالي:

```
node
```

ستلاحظ ظهور الرمز > في بداية السطر عند الدخول إلى وضع REPL التفاعلي، ويمكننا إدخال شيفرات جافاسكربت لتنفيذها كالتالي:

```
const Todos = require('./index');
```

نحمل الوحدة البرمجية لمدير قائمة المهام باستخدام التابع `require()` ونخزن قيمتها ضمن متغير بالاسم `Todos`، والذي صدرنا منه افتراضيًا الصنف `Todos`، والآن لنبدأ بإنشاء كائن جديد من ذلك الصنف كالتالي:

```
const todos = new Todos();
```

يمكننا اختبار الوحدة باستخدام الكائن `todos` المشتق من الصنف `Todos` للتأكد من عمله وفق ما هو متوقع، فنبدأ بإضافة مهمة جديدة كالتالي:

```
todos.add("run code");
```

لم يظهر إلى الآن مما نفذناه أي خرج ضمن الطرفية، ولنتأكد من تخزين المهمة السابقة بشكل سليم ضمن قائمة المهام نستدعي تابع عرض المهام الموجودة ونعاين النتيجة:

```
todos.list();
```

سيظهر لنا الخرج التالي:

```
[ { title: 'run code', completed: false } ]
```

وهي النتيجة الصحيحة المتوقعة حيث تحتوي على عنصر وحيد وهو المصفوفة التي أضفناها سابقًا وحالة اكتمالها غير مكتملة، لنضيف الآن مهمة أخرى ونعدل المهمة الأولى لتصبح مكتملة كالتالي:

```
todos.add("test everything");
todos.complete("run code");
```

نتوقع الآن وجود مهمتين ضمن الكائن `todos`، وهما "run code" و "test everything"، حيث يجب أن تكون المهمة الأولى "run code" مكتملة، ونتأكد من ذلك باستدعاء التابع `list()`:

```
todos.list();
```

نحصل على الخرج:

```
[
  { title: 'run code', completed: true },
  { title: 'test everything', completed: false }
]
```

الخرج صحيح كما هو متوقع، والآن نخرج من جلسة REPL بتنفيذ الأمر التالي:

```
.exit
```

بذلك نكون قد تحققنا من عمل الوحدة البرمجية التي طورناها بشكل سليم، ولم نستخدم في ذلك أي ملفات اختبار مخصصة أو مكتبات اختبار، بل اعتمدنا فقط على الاختبار اليدوي، ولكن هذه الطريقة في الاختبار تأخذ وقتًا وجهدًا في كل مرة نضيف فيها تعديلات على الوحدة البرمجية، لذا سنؤتمت في الفقرة التالية عملية الاختبار هذه ونرى ما يمكن لإطار العمل موكا أن يساعدنا في ذلك.

6.3 كتابة اختبارات Node.js باستخدام Mocha و Assert

اختبرنا في الفقرة السابقة التطبيق يدويًا مع أن ذلك قد يفيد في بعض الحالات إلا أنه ومع تطوير الوحدة البرمجية التي نعمل عليها وزيادة حجمها والشيفرات المستخدمة ضمنها ستزداد تلك الطريقة صعوبة، وبينما نحن نختبر المزايا الجديدة التي أضفناها فيجب أيضًا اختبار المزايا السابقة جميعها مجددًا للتأكد أن التطويرات لم تؤثر على أي مزايا سابقة، وسيجبرنا ذلك اختبار كل ميزة ضمن التطبيق مرارًا وتكرارًا في كل مرة نعدل فيها الشيفرة ما سيأخذ الكثير من الوقت والجهد وقد نخطئ أو ننسى تنفيذ بعض الاختبارات خلال تلك المرحلة.

والحل إعداد اختبارات مؤتمتة عبارة عن نصوص اختبار برمجية كأي برنامج عادي آخر، نمرر فيها بيانات محددة إلى التوابع ضمن التطبيق ونتأكد من سلامة عملها ووظيفتها كما هو متوقع، وكلما أضفنا ميزة للتطبيق أضفنا معها اختبارها، حيث عندما نكتب اختبارات مقابلة لكل ميزة في التطبيق سنتحقق بذلك من عمل الوحدة البرمجية كاملةً ودون حاجة لتذكر تنفيذ كل التوابع واختبار كل المزايا في كل عملية اختبار.

وسنستخدم في كتابة الاختبارات إطار عمل مخصص للاختبار يدعى موكا Mocha مع وحدة `assert` البرمجية التي يوفرها نود كما أشرنا في بداية الفصل، والآن نبدأ بإنشاء ملف جديد سنضع داخله شيفرات الاختبار كالتالي:

```
touch index.test.js
```

نفتحه ضمن أي محرر نصوص:

```
nano index.test.js
```

نبدأ بتحميل الوحدة البرمجية لمدير المهام كما فعلنا ضمن جلسة REPL في الفقرة السابقة، وبعدها نحمل الوحدة البرمجية `assert` لاستخدامها عند كتابة الاختبارات كالتالي:

```
const Todos = require('./index');
const assert = require('assert').strict;
```

تسمح الخاصية `strict` التي استخرجناها من الوحدة `assert` باستخدام معامل مساواة خاص منصوح باستخدامه ضمن بيئة نود ويوفر مزايا مفيدة أخرى لن ندخل في تفاصيلها.

والآن قبل كتابة الاختبارات لنتعرف على طريقة موكا Mocha في تنظيم وترتيب شيفرات الاختبار، حيث تُكتب الاختبارات في Mocha بالصيغة التالية:

```
describe([Test Group Name], function() {
    it([Test Name], function() {
        [Test Code]
    });
});
```

لاحظ وجود استدعاء لدالتين رئيسيين هما `describe()` و `it()`، حيث تستخدم الدالة `describe()` لتجميع الاختبارات المتشابهة معًا المكتوبة عبر `it()`، ولكن خطوة التجميع هذه غير ضرورية لكنها تسهل قراءة ملفات الاختبار وتزيد تنظيمها ويسهل لاحقًا التعديل على الاختبارات المتشابهة بسهولة أكبر، أما الدالة `it()` فتحتوي على شيفرة الاختبار المراد تنفيذها للوحدة البرمجية المختبرة ونستخدم فيها مكتبة `assert` للتوكيد والتحقق من المخرجات.

وهدفنا في هذه الفقرة استخدام موكا Mocha والوحدة `assert` لأتمتة عملية الاختبار أو حتى تنفيذها يدويًا كما فعلنا سابقًا، لذلك سنبدأ أولًا بتعريف مجموعة اختبارات باستخدام التابع `describe()` بإضافة الأسطر التالية لملف الاختبار بعد استيراد الوحدات البرمجية السابقة:

```
...
describe("integration test", function() {
});
```

بهذا نكون قد أنشأنا مجموعة اختبارات -سنكتبها لاحقًا- باسم integration test أي اختبار التكامل ووظيفته التحقق من عمل عدة توابع مع بعضها ضمن الوحدات البرمجية، على عكس اختبار الوحدة unit test الذي يختبر دالة واحدة في كل مرة، وعندما ينفذ موكا عملية اختبار التطبيق فسينفذ كل الاختبارات المعرفة ضمن التابع `describe()` ضمن مجموعة بالاسم "integration test" التي عرفناها.

والآن لنضيف اختبارًا باستخدام التابع `it()` لاختبار جزء من التطبيق:

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function() {
    });
});
```

نلاحظ كيف سمينا الاختبار باسم أجنبي واضح يصف عمله معناه بالعربية "يجب التمكن من إضافة مهمة ToDo وإكمالها"، لذا عند تنفيذ أي شخص للاختبارات سيعرف ما الجزء الذي نجح في الاختبار من تلك التي لم تنجح فيه، حيث يعتبر الاختبار الجيد لأي تطبيق توثيقًا جيدًا لعمله فتعتبر تلك الاختبارات كتوثيق تقني للتطبيق.

والآن نبدأ بأول اختبار وهو إنشاء كائن من الصنف `Todos` جديد والتأكد بأنه لا يحتوي على أي عناصر:

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function() {
        let todos = new Todos();
        assert.notStrictEqual(todos.list().length, 1);
    });
});
```

أنشأنا في أول سطر من الاختبار كائنًا جديدًا من الصنف `Todos` كما فعلنا سابقًا ضمن REPL أو كما سنفعل عند استخدام هذا الصنف ضمن أي وحدة برمجية أخرى، واستخدمنا في السطر الثاني الوحدة `assert` وتحديدًا تابع اختبار عدم المساواة `notStrictEqual()` والذي يأخذ معاملان وهما القيمة التي نريد اختبارها وتدعى القيمة الفعلية `actual`، والمعامل الثاني وهو القيمة التي نتوقع أن لا تساويها وتدعى القيمة المتوقعة `expected`، وفي حال تساوي القيمتين سيرمي التابع `notStrictEqual()` خطئًا ويفشل هذا الاختبار.

نحفظ الملف ونخرج منه، ونتوقع في هذه الحالة نجاح هذا الاختبار لأن طول المصفوفة سيكون `0` وهو غير مساوي للقيمة `1`، ونتأكد من ذلك بتشغيل الاختبارات باستخدام موكا، لذا نعدل بدايةً على ملف الحزمة package.json ونفتحه ضمن محرر النصوص ونعدل النص البرمجي الخاص بتشغيل الاختبارات ضمن الخاصية `scripts` كالتالي:

```
...
"scripts": {
    "test": "mocha index.test.js"
},
...
```

بذلك نكون قد عدلنا الأمر `test` الخاص بـالأداة npm، حيث عنـد تنفيـذه كالتـالي `npm test` سـيتحقق npm من الأمر الذي أدخلناه ضمن ملـف الحزمـة package.json وسـيبحث عن مكتبـة موكـا Mocha ضـمن مجلد الحزم node_modules وينفذ الأمر `mocha` مُمرِّرًا له اسم ملف الاختبار للتطبيق.

والآن نحفظ الملف ونخرج منه وننفذ أمر الاختبار السابق ونعاين النتيجة كالتالي:

```
npm test
```

نحصل على الخرج:

```
> todos@1.0.0 test your_file_path/todos
> mocha index.test.js



integrated test
    ✓ should be able to add and complete TODOs



passing (16ms)
```

يُظهر لنا الخرج السابق مجموعة الاختبارات التي جرى تنفيذها ويترك فراغًا قبل كل اختبـار ضـمن مجموعـة الاختبار المعرفة، ونلاحظ ظهور اسم الاختبـار كمـا مررنـاه للتـابع `it()` في ملـف الاختبـار، حيث تشـير العلامـة الظاهرة على يسار الاختبار أن الاختبار قد نجح، وفي الأسفل يظهر لنا خلاصة فيها معلومـات عن كـل الاختبـارات التي نُفِّذت، وفي حالتنا هناك اختبار واحد ناجح واستغرقت عملية الاختبار كاملة 16 ميلي ثانية لتنفيـذها، حيث يعتمد هذا التوقيت على أداء الجهاز الذي يُنفذ تلك الاختبارات.

وكما لاحظنا أن الاختبارات التي نفذناها نجحت بالكامل ولكن مع ذلك فإن الاختبار الذي كتبناه قد يشير إلى حالة نجاح مغلوطة، وهي الحالة التي ينجح فيها الاختبار بينما في الحقيقـة يجب أن يفشـل، حيث أننـا في هـذا الاختبار نختبر فراغ مصفوفة المهام باختبار أن طولها لا يساوي الواحد، لذا نعـدل الاختبـار السـابق ضـمن ملـف الاختبارات index.test.js ليصبح كالتالي:

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function() {
        let todos = new Todos();
        todos.add("get up from bed");
        todos.add("make up bed");
```

```
        assert.notStrictEqual(todos.list().length, 1);
    });
});
```

نحفظ الملف ونخرج منه ونلاحظ أننا أضفنا مهمتين جديدتين، لننفذ الاختبار ونلاحظ النتيجة:

```
npm test
```

سيظهر لنا التالي:

```
...
integrated test
    ✓ should be able to add and complete TODOs


passing (8ms)
```

نرى أن الاختبار قد نجح لأن طول المصفوفة ليس واحد كما هو متوقع، لكن ذلك يتعارض مع الاختبار السابق الذي أجريناه، حيث مهمته التحقق من أن الكائن الجديد من مدير المهام سيبدأ فارغًا دون أي مهام مخزنة ضمنه، لذا من الأفضل أن يتحقق الاختبار من ذلك في جميع الحالات.

لنعدل الاختبار ونجعله ينجح فقط في حال عدم وجود أي مهام مخزنة ضمن الكائن ليصبح كالتالي:

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function() {
        let todos = new Todos();
        todos.add("get up from bed");
        todos.add("make up bed");
        assert.strictEqual(todos.list().length, 0);
    });
});
```

لاحظ استدعينا التابع `strictEqual()` بدلًا من استدعاء التابع `notStrictEqual()` الذي يتحقق من المساواة بين القيمة الحقيقية والمتوقعة الممررة له، بحيث يفشل عند عدم تساوي القيمتين.

والآن نحفظ الملف ونخرج منه ونعيد تنفيذ أمر الاختبار لنرى النتيجة:

```
npm test
```

هذه المرة سيظهر لنا خطأ:

```
...
  integration test
should be able to add and complete TODOs



passing (16ms)
failing

integration test
       should be able to add and complete TODOs:

      AssertionError [ERR_ASSERTION]: Input A expected to strictly
equal input B:
+ expected - actual

- 2
+ 0
      + expected - actual

      -2
      +0

      at Context. (index.test.js:9:10)



npm ERR! Test failed.  See above for more details.
```

سيفيدنا الخرج الظاهر في معرفة سبب الفشل وتصحيح الخطأ الحاصل، ونلاحظ عدم وجود علامة بجانب اسم الاختبار لأنه فشل، وأيضًا لم تعد خلاصة تنفيذ عملية الاختبار في الأسفل بل في الأعلى بعد قائمة الاختبارات المنفذة وحالتها:

```
...
passing (29ms)
failing
...
```

والخرج الباقي يظهر بيانات متعلقة بالاختبارات الفاشلة، حيث يظهر أولًا الاختبارات التي فشلت:

```
...
integrated test
       should be able to add and complete TODOs:
...
```

ثم سبب فشل تلك الاختبارات:

```
...
      AssertionError [ERR_ASSERTION]: Input A expected to strictly
equal input B:
+ expected - actual

- 2
+ 0
      + expected - actual

      -2
      +0

      at Context. (index.test.js:9:10)
...
```

رُمي خطأ من النوع `AssertionError` عندما فشل اختبار التابع `strictEqual()`، حيث نلاحظ أن القيمة المتوقعة وهي `0` مختلفة عن القيمة الحقيقية لطول مصفوفة المهام وهي `2`، ونلاحظ ذكر السطر الذي فشل عنده الاختبار ضمن ملف الاختبار وهو السطر رقم 10، وتفيد هذه المعلومات في حل المشكلة.

نعدل الاختبار ونصحح المشكلة بتوقع القيمة الصحيحة لطول المصفوفة حتى لا يفشل الاختبار، وأولًا نفتح ملف الاختبارات:

```
nano index.test.js
```

ثم نزيل أسطر إضافة المهام باستخدام `todos.add` ليصبح الاختبار كالتالي:

```
...
describe("integration test", function () {
    it("should be able to add and complete TODOs", function () {
        let todos = new Todos();
```

```
        assert.strictEqual(todos.list().length, 0);
    });
});
```

نحفظ الملف ونخرج منه ونعيد تنفيذ الاختبار مجددًا ونتأكد من نجاحه في حالة صحيحة هذه المرة وليست مغلوطة:

```
npm test
```

نحصل على الخرج:

```
...
integration test
    ✓ should be able to add and complete TODOs


passing (15ms)
```

أصبح الاختبار الآن أقرب لما نريد، لنعود إلى اختبار التكامل مجددًا ونحاول اختبار إضافة مهمة جديدة ضمن الملف index.test.js كالتالي:

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function() {
        let todos = new Todos();
        assert.strictEqual(todos.list().length, 0);

        todos.add("run code");
        assert.strictEqual(todos.list().length, 1);
        assert.deepStrictEqual(todos.list(), [{title: "run code",
completed: false}]);
    });
});
```

بعد استدعاء التابع `add()` نتحقق من وجود مهمة واحدة ضمن كائن مدير المهام `todos` باستخدام تابع التوكيد `strictEqual()`، وأما الاختبار التالي فسيتحقق من البيانات الموجودة ضمن قائمة المهام `todos` بواسطة التابع `deepStrictEqual()` والذي يختبر مساواة القيمة المتوقعة مع القيمة الحقيقية تعاوديًا بالمرور على كل الخصائص ضمن من القيمتين واختبار مساواتهما، ففي حالتنا سيختبر أن المصفوفتين يملك

كل منها كائنًا واحدًا داخلها، ويتحقق من امتلاك كلا الكائنين لنفس الخواص وتساويها ففي حالتنا يجب أن يكون هنالـك خاصـيتين الأولى العنـوان `title` ويجب أن تسـاوي قيمتهـا `"run code"` والثـاني اكتمـال المهمـة `completed` وقيمتها تساوي `false`.

نكمل كتابة الاختبار ليصبح كالتالي:

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function() {
        let todos = new Todos();
        assert.strictEqual(todos.list().length, 0);

        todos.add("run code");
        assert.strictEqual(todos.list().length, 1);
        assert.deepStrictEqual(todos.list(), [{title: "run code",
completed: false}]);

        todos.add("test everything");
        assert.strictEqual(todos.list().length, 2);
        assert.deepStrictEqual(todos.list(),
            [
                { title: "run code", completed: false },
                { title: "test everything", completed: false }
            ]
        );

        todos.complete("run code");
        assert.deepStrictEqual(todos.list(),
            [
                { title: "run code", completed: true },
                { title: "test everything", completed: false }
            ]
    );
  });
});
```

أصبح الاختبار الآن مماثل تمامًا للاختبار اليدوي الذي نفذناها سابقًا، ولم نعد بحاجة للتحقق من الخرج يدويًا في كل مرة، فيكفي تنفيذ هذا الاختبار والتأكد من نجاحه ليدل على صحة عمل الوحدة البرمجية، حيث الهدف من الاختبار في النهاية التأكد من سلامة عمل الوحدة البرمجية كلها.

والآن نحفظ الملف ونخرج منه وننفذ الاختبارات مرة أخرى ونتحقق من النتيجة:

```
...
integrated test
    ✓ should be able to add and complete TODOs



passing (9ms)
```

أعددنا بذلك اختبار تكامل باستخدام إطار الاختبارات موكا Mocha والوحدة `assert`.

والآن لنتخيل بأننا شاركنا الوحدة البرمجية السابقة مع مطورين آخرين وأخبرنا العديد منهم بأنه يفضل رمي خطأ عند استدعاء التابع `complete()` في حال لم يتم إضافة أي مهام بعد سابقًا، لذا لنضيف تلك الخاصية ضمن التابع `complete()` ضمن الملف index.js:

```
...
complete(title) {
    if (this.todos.length === 0) {
        throw new Error("You have no TODOs stored. Why don't you add
one first?");
    }

    let todoFound = false
    this.todos.forEach((todo) => {
        if (todo.title === title) {
            todo.completed = true;
            todoFound = true;
            return;
        }
    });

    if (!todoFound) {
        throw new Error(`No TODO was found with the title: "$
{title}"`);
```

```
    }
}
...
```

نحفظ الملف ونخرج منه ثم نضيف اختبارًا جديدًا لتلك الميزة في ملف الاختبارات، حيث نريد التحقق من أن استدعاء التابع `complete` من كائن لا يحوي أي مهام بعد سيعيد الخطأ الخاص بحالتنا، لذا نعود لملف الاختبار ونضيف في نهايته الشيفرة التالية:

```
...
describe("complete()", function() {
    it("should fail if there are no TODOs", function() {
        let todos = new Todos();
        const expectedError = new Error("You have no TODOs stored. Why
don't you add one first?");

        assert.throws(() => {
            todos.complete("doesn't exist");
        }, expectedError);
    });
});
```

استخدما التوابع `describe()` و `it()` كما فعلنا سابقًا، وبدأنا الاختبار بإنشاء كائن `todos` جديد، ثم عرّفنا الخطأ المتوقع عند استدعاء التابع `complete()` واستخدمنا تابع توكيد رمي الأخطاء `throws()` الذي توفره الوحدة `assert` لاختبار الأخطاء المرمية من قبل الشيفرة عند تنفيذها، حيث نمرر له كمعامل أول تابعًا يحتوي داخله على التابع الذي نتوقع منه رمي الخطأ، والمعامل الثاني هو الخطأ المتوقع رميه، والآن ننفذ أمر الاختبار `npm test` ونعاين النتيجة:

```
...
integrated test
    ✓ should be able to add and complete TODOs

  complete()
    ✓ should fail if there are no TODOs


passing (25ms)
```

يتضح من الخرج السابق أهمية أتمتة الاختبارات باستخدام موكا والوحدة assert، حيث أنه وعند كل تنفيذ لأمر الاختبار npm test سيتم التحقق من نجاح كل الاختبارات السابقة، ولا حاجة لتكرار التحقق اليدوي أبدًا طالما أن الشيفرات الأخرى لا زالت تعمل وهذا ما تأكدنا منه عند نجاح بقية الاختبارات.

وإلى الآن كل ما اختبرناه كان عبارة عن شيفرات متزامنة، وفي الفقرة التالية سنتعلم طرق اختبار والتعامل مع الشيفرات اللامتزامنة.

6.4 اختبار الشيفرات اللامتزامنة

سنضيف الآن ميزة تصدير قائمة المهام إلى ملف بصيغة CSV التي ذكرناها سابقًا، حيث سيحوي ذلك الملف كل المهام المخزنة مع تفاصيل حالة اكتمالها، لذا وللتعامل مع نظام الملفات سنحتاج لاستخدام وحدة fs التي توفرها نود لكتابة ذلك الملف.

والجدير بالذكر أن عملية كتابة الملف عملية غير متزامنة ويمكن تنفيذها بعدة طرق كاستخدام دوال رد النداء callbacks مثلًا أو الوعود Promises أو عبر اللاتزامن والانتظار async/await كما رأينا في الفصل السابق.

سنتعلم في هذه الفقرة كيف يمكن كتابة الاختبارات للشيفرات اللامتزامنة التي تستخدم أي طريقة من تلك الطرق.

6.4.1 الاختبار باستخدام دوال رد النداء

تمُرر دالة رد النداء كمعامل إلى التابع اللامتزامن لتُستدعى عند انتهاء مهمة ذلك التابع، لنبدأ بإضافة التابع saveToFile() للصنف Todos والذي سيمر على عناصر المهام ضمن الصنف ويبنى منها سلسلة نصية ويخزنها ضمن ملف بصيغة CSV، لذا نعود إلى ملف index.js ونضيف الشيفرات المكتوبة في نهايته:

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

class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }
    add(title) {
        let todo = {
            title: title,
```

```
            completed: false,
        }
        this.todos.push(todo);
    }

    complete(title) {
        if (this.todos.length === 0) {
            throw new Error("You have no TODOs stored. Why don't you
add one first?");
        }

        let todoFound = false
        this.todos.forEach((todo) => {
            if (todo.title === title) {
                todo.completed = true;
                todoFound = true;
                return;
            }
        });

        if (!todoFound) {
            throw new Error(`No TODO was found with the title: "$
{title}"`);
        }
    }

    saveToFile(callback) {
        let fileContents = 'Title,Completed\n';
        this.todos.forEach((todo) => {
            fileContents += `${todo.title},${todo.completed}\n`
        });

        fs.writeFile('todos.csv', fileContents, callback);
    }
}
module.exports = Todos;
```

بداية نستورد الوحدة `fs` ثم نضيف التابع الجديد `saveToFile()` إلى الصنف والذي يقبل كمعامل له دالة رد نداء تُستدعَى عند انتهاء عملية كتابة الملف، ونُنشئ ضمن التابع الجديد محتوى الملف ونخزنه ضمن المتغير `fileContents`، ونلاحظ كيف عيّنا القيمة الابتدائية له وهي عناوين الأعمدة للجدول في ملف CSV، ومررنا على كل مهمة مخزنة ضمن المصفوفة باستخدام التابع `forEach()` وأضفنا لكل مهمة قيمة خاصية العنوان لها `title` وحالة الاكتمال `completed`، ثم استدعينا التابع `writeFile()` من وحدة `fs` لكتابة الملف النهائي، ومررنا له اسم الملف الناتج todos.csv وكمعامل ثانٍ مررنا محتوى ذلك الملف وهو قيمة المتغير `fileContents` السابق، وآخر معامل هو دالة رد النداء لمعالجة الخطأ الذي قد يحدث عند تنفيذ هذه العملية.

والآن نحفظ الملف ونخرج منه ونكتب اختبارًا للتابع الجديد `saveToFile()` يتحقق من وجود الملف المصدَّر ثم يتحقق من صحة محتواه، لذا نعود لملف الاختبار index.test.js ونبدأ بتحميل الوحدة `fs` في بداية الملف والتي سنستخدمها في عملية الاختبار:

```
const Todos = require('./index');
const assert = require('assert').strict;
const fs = require('fs');
...
```

ونضيف حالة الاختبار الجديدة في نهاية الملف:

```
...
describe("saveToFile()", function() {
    it("should save a single TODO", function(done) {
        let todos = new Todos();
        todos.add("save a CSV");
        todos.saveToFile((err) => {
            assert.strictEqual(fs.existsSync('todos.csv'), true);
            let expectedFileContents = "Title,Completed\nsave a
CSV,false\n";
            let content = fs.readFileSync("todos.csv").toString();
            assert.strictEqual(content, expectedFileContents);
            done(err);
        });
    });
});
```

وبما أن هذا اختبار لميزة جديدة كليًا عن سابقاتها استخدمنا الدالة `describe()` لتعريف مجموعة اختبارات جديدة متعلقة بهذه الميزة، ونلاحظ هذه المرة استخدام الدالة `it()` بطريقة مختلفة، حيث نمرر لها

عادةً دالة رد نداء تحوي داخلها الاختبار دون تمرير أي معامل لها، ولكن هذه المرة سنمرر لدالة رد النداء المعامل done والذي نحتاج إليه عند تنفيذ اختبار شيفرات لا متزامنة وخصوصًا التي تستخدم في عملها دوال رد النداء، حيث نستخدم دالة رد النداء done() لإعلام موكا عند الانتهاء من اختبار عملية غير متزامنة، لهذا يجب علينا التأكد بعد اختبار دوال رد النداء استدعاء done() حيث بدون ذلك الاستدعاء لن يعلم موكا أن الاختبار قد انتهى وسيبقى منتظرًا إشارة الانتهاء تلك.

ونلاحظ أننا أنشأنا كائنًا جديدًا من الصنف Todos وأضفنا مهمة جديدة له بعدها استدعينا التابع الجديد saveToFile() ومررنا له دالة لفحص كائن الخطأ الذي سيمرر لها إن وجد، ونلاحظ كيف وضعنا الاختبار لهذا التابع ضمن دالة رد النداء، لأن الاختبار سيفشل إن أجري قبل عملية كتابة الملف.

أول توكيد تحققنا منه هو أن الملف todos.csv موجود:

```
...
assert.strictEqual(fs.existsSync('todos.csv'), true);
...
```

حيث يعيد التابع fs.existsSync() القيمة الصحيحة true إذا كان الملف المحدد بالمسار الممرر له موجودًا وإلا سيعيد قيمة خاطئة false.

> يرجع العمل أن توابع الوحدة fs غير متزامنة افتراضيًا، ويوجد لبعض التوابع الأساسية منها نسخ متزامنة استخدمناها هنا لتبسيط الاختبار ويمكننا الاستدلال على تلك التوابع المتزامنة من اللاحقة "Sync" في نهاية اسمها، فلو استخدمنا النسخة المتزامنة ومررنا لها دالة رد نداء أيضًا فستصبح الشيفرة متداخلة وصعبة القراءة أو التعديل.

أنشأنا بعد ذلك متغيرًا يحوي القيمة المتوقعة للملف todos.csv:

```
...
let expectedFileContents = "Title,Completed\nsave a CSV,false\n";
...
```

واستخدمنا التابع المتزامن readFileSync() من الوحدة fs لقراءة محتوى الملف كالتالي:

```
...
let content = fs.readFileSync("todos.csv").toString();
...
```

حيث مررنا للتابع readFileSync() مسار الملف todos.csv الذي جرى تصديره، وسيعيد لنا كائن تخزين مؤقت Buffer سيحوي بيانات الملف بالصيغة الثنائية، لذا نستدعي التابع toString() منه

للحصول على القيمة النصية المقروءة لتلك البيانات لمقارنتها مع القيمة المتوقعة لمحتوى الملف التي أنشأناها مسبقًا، ونستخدم لمقارنتهما تابع اختبار المساواة `strictEqual` من الوحدة `assert`:

```
...
assert.strictEqual(content, expectedFileContents);
...
```

وأخيرًا نستدعي التابع `done()` لإعلام موكا بانتهاء الاختبار:

```
...
done(err);
...
```

نلاحظ كيف مررنا كائن الخطأ `err` عند استدعاء تابع الانتهاء `done()` حيث سيفحص موكا تلك القيمة وسيفشل الاختبار إن احتوت على خطأ.

والآن نحفظ الملف ونخرج منه وننفذ الاختبارات بتنفيذ التابع `npm test` كما العادة ونلاحظ النتيجة:

```
...
integrated test
    ✓ should be able to add and complete TODOs

  complete()
    ✓ should fail if there are no TODOs

  saveToFile()
    ✓ should save a single TODO


passing (15ms)
```

بذلك نكون قد اختبرنا تابعًا غير متزامنًا يستخدم دالة رد النداء، وبما أن تلك الطريقة لم تعد مستخدمة كثيرًا في وقتنا الحالي وتم استبدالها باستخدام الوعود كما شرحنا في الفصل الخامس من هذا الكتاب، سنتعلم في الفقرة القادمة كيف يمكن اختبار الشيفرات التي تستخدم الوعود في تنفيذ عملياتها اللامتزامنة باستخدام موكا.

6.4.2 الاختبار باستخدام الوعود

الوعد Promise هو كائن توفره جافاسكربت وظيفته إرجاع قيمة ما لاحقًا، وعندما تنفذ عمليته بنجاح نقول تحقق ذلك الوعد resolved، وفي حال حدث خطأ في تنفيذ عمليته نقول أنه قد فشل rejected.

لنبدأ بتعديل التابع `saveToFile()` ليستخدم الوعود بدلًا من دوال رد النـداء، نفتح ملـف index.js ونبـدأ بتعديل طريقة استيراد الوحدة `fs`، حيث نعدل على عبارة الاستيراد باستخدام `require()` لتصبح كالتالي:

```
...
const fs = require('fs').promises;
...
```

بذلك نكون قد استوردنا وحدة `fs` التي تستخدم الوعود بدلًا من التي تستخدم دوال رد النداء، ثم نعدل التابع `saveToFile()` ليستخدم الوعود بشكل سليم كالتالي:

```
...
saveToFile() {
    let fileContents = 'Title,Completed\n';
    this.todos.forEach((todo) => {
        fileContents += `${todo.title},${todo.completed}\n`
    });

    return fs.writeFile('todos.csv', fileContents);
}
...
```

نلاحظ أن التابع لم يعد يقبل معاملًا له، حيث يغنينا استخدام الوعود عن ذلك، ونلاحـظ أيضًـا تغيـير طريقـة كتابة التابع حيث نرجع منه الوعد الذي يرجعه التابع `writeFile()`.

والآن نحفظ التغييرات على ملف index.js ثم نعدل على اختبار هذا التابع ليلائم استخدامه للوعود، لذا نعود لملف الاختبار index.test.js ونبدل اختبار التابع `saveToFile()` ليصبح كالتالي:

```
...
describe("saveToFile()", function() {
    it("should save a single TODO", function() {
        let todos = new Todos();
        todos.add("save a CSV");
        return todos.saveToFile().then(() => {
            assert.strictEqual(fs.existsSync('todos.csv'), true);
            let expectedFileContents = "Title,Completed\nsave a
CSV,false\n";
            let content = fs.readFileSync("todos.csv").toString();
            assert.strictEqual(content, expectedFileContents);
```

```
        });
    });
});
```

أول تعديل أجريناه هو إزالة معامل تابع الانتهاء `done()` لأن بقاءه يعني انتظار موكا إشارة استدعائه حتى ينهي الاختبار وإلا سيرمي خطأ كالتالي:

```
saveToFile()
       should save a single TODO:
     Error: Timeout of 2000ms exceeded. For async tests and hooks,
ensure "done()" is called; if returning a Promise, ensure it resolves.
(/home/ubuntu/todos/index.test.js)
      at listOnTimeout (internal/timers.js:536:17)
      at processTimers (internal/timers.js:480:7)
```

لهذا السبب عندما نستخدم الوعود ضمن الاختبار لا نمرر المعامل `done()` إلى دالة رد النداء المُمررة لدالة تعريف الاختبار `it()`.

ولاختبار الوعد نضع اختبارات التوكيدات ضمن استدعاء التابع `then()`، ونلاحظ كيف أننا نرجع الوعد من داخل تابع الاختبار وأننا لا نضيف استدعاء للتابع `catch()` إليه لالتقاط الخطأ الذي قد يُرمى أثناء التنفيذ، وذلك حتى تصل أي أخطاء ترمى من داخل التابع `then()` إلى الدالة الأعلى وتحديدًا إلى `it()`، حتى يعلم موكا بحدوث أخطاء أثناء التنفيذ وإفشال الاختبار الحالي، لذلك ولاختبار الوعود يجب أن نعيد الوعد المراد اختباره باستخدام `return`، وإلا سيظهر الاختبار على أنه ناجح حتى عند فشله في الحقيقة، ونحصل على نتيجة صحة مغلوطة، وأيضًا نتجاهل إضافة التابع `catch()` لأن موكا يتحقق من الأخطاء المرمية بنفسه للتأكد من حالة فشل الوعد الذي يجب أن يؤدي بالمقابل إلى فشل الاختبار الذي يعطينا فكرة عن وجود مشكلة في عمل وحدة التطبيق.

والآن وبعد أن عدلنا الاختبار نحفظ الملف ونخرج منه، وننفذ الأمر `npm test` لتنفيذ الاختبارات والتأكد من نجاحها:

```
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs
```

```
  saveToFile()
    ✓ should save a single TODO


passing (18ms)
```

بذلك نكون قد عدلنا على الشيفرة والاختبار المتعلق بها لتستخدم الوعود، وتأكدنا من أن الميزة لا زالت تعمل بشكل صحيح، والآن بدلًا من التعامل مع الوعود بتلك الطريقة سنستخدم في الفقرة التالية اللاتزامن والانتظار `async/await` لتبسيط العملية وإلغاء الحاجة لاستدعاء التابع `then()` أكثر من مرة لمعالجة حالات نجاح التنفيذ ولتبسيط شيفرة الاختبار وتوضيحها.

6.4.3 الاختبار باستخدام اللاتزامن والانتظار async/await

تتيح الكلمتان المفتاحيتان `async/await` صيغة بديلة للتعامل مع الوعود، فعند تحديد تابع ما كتابع لا متزامن باستخدام الكلمة المفتاحية `async` يصبح بإمكاننا الحصول داخله مباشرةً على قيمة نتيجة أي وعد ننفذه عند نجاحه باستخدام الكلمة المفتاحية `await` قبل استدعاء الوعد، وبذلك نلغي الحاجة لاستدعاء التوابع `then()` أو `catch()` نهائيًا، وباستخدامها يمكننا تبسيط اختبار التابع `saveToFile()` الذي يستخدم الوعود، لذا نعدله ضمن ملف الاختبارات index.test.js ليصبح كالتالي:

```
...
describe("saveToFile()", function() {
    it("should save a single TODO", async function() {
        let todos = new Todos();
        todos.add("save a CSV");
        await todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CSV,false\
n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });
});
```

نلاحظ كيف أضفنا الكلمة `async` قبل تعريف دالة رد النداء المُمرر إلى `it()`، ما يسمح لنا باستخدام الكلمة `await` داخلها، ونلاحظ عند استدعاء التابع `saveToFile()` إضافة الكلمة `await` قبل استدعائه بذلك لن يكمل نود تنفيذ الشيفرات في الأسطر اللاحقة وسينتظر لحين انتهاء تنفيذ هذا التابع، ونلاحظ أيضًا كيف

أصبحت شيفرة الاختبار أسهل في القراءة بعد أن نقلنا شيفرات التوكيد من داخل التابع `then()` مباشرة إلى جسم تابع الاختبار المُمرر إلى `it()`.

والآن ننفذ الاختبارات بتنفيذ الأمر `npm test` لنحصل على الخرج:

```
...
integrated test
    ✓ should be able to add and complete TODOs

  complete()
    ✓ should fail if there are no TODOs

  saveToFile()
    ✓ should save a single TODO


passing (30ms)
```

بذلك نكون قد تعلمنا كيف يمكن اختبار التوابع اللامتزامنة مهما كان شكلها، سواء كانت تستخدم دوال رد النداء في عملها أو الوعود، وتعلمنا الكثير عن إطار عمل الاختبار موكا Mocha وكيفية استخدامه لاختبار التوابع اللامتزامنة، وسنتعرف في الفقرة التالية على خصائص أخرى يوفرها موكا Mocha ستحسن من طريقة كتابة الاختبارات، وسنتعرف تحديدًا على الخطافات hooks وكيف يمكنها التعديل على بيئة الاختبار.

6.5 تحسين الاختبارات باستخدام الخطافات Hooks

تسمح الخطافات في موكا Mocha بإعداد بيئة الاختبار قبل وبعد تنفيذ الاختبارات، حيث نستخدمها داخل التابع `describe()` عادةً وتحوي على شيفرات تفيد في عملية الإعداد والتنظيف التي قد تحتاجها بعض الاختبارات، حيث يوفر موكا أربع خطافات رئيسية وهي:

- `before`: يُنفذ مرة واحدة قبل أول اختبار فقط.
- `beforeEach`: يُنفذ قبل كل اختبار.
- `after`: يُنفذ بعد تنفيذ آخر اختبار فقط.
- `afterEach`: يُنفذ بعد كل اختبار.

تفيد تلك الخطافات عند اختبار تابع ما ضمن عدة اختبارات، وتسمح بفصل شيفرة الإعداد لها إلى مكان واحد منفصل عن مكان شيفرات التوكيد، كإنشاء الكائن `todos` في حالتنا مثلًا، ولنختبر فائدتها سنبدأ أولًا

بإضافة اختبارات جديدة لمجموعة اختبارات التابع `saveToFile()`، فبعد أن تحققنا في الاختبـار الماضـي من صحة تصدير ملف المهام إلا أننا اختبرنا وجود مهمة واحدة فقط ضمنه، ولم نختبر الحالة التي تكون فيها المهمـة مكتملة وهل سيتم حفظها ضمن الملف بشكل سليم أم لا، لـذلك سنضـيف اختبـارات جديـدة للتأكـد من تلـك الحالات وبالتالي التأكد من صحة عمل الوحدة البرمجية التي نطورها.

لنبدأ بإضافة اختبار ثانٍ للتأكـد من حفـظ المهـام المكتملـة بشـكل سـليم، لـذا نفتح الملـف index.test.js ضمن محرر النصوص ونضيف الاختبار الجديد كالتالي:

```
...
describe("saveToFile()", function () {
    it("should save a single TODO", async function () {
        let todos = new Todos();
        todos.add("save a CSV");
        await todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CSV,false\
n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });

    it("should save a single TODO that's completed", async function ()
{
        let todos = new Todos();
        todos.add("save a CSV");
        todos.complete("save a CSV");
        await todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CSV,true\
n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });
});
```

يشبه هذا الاختبار ما سبقه، والفرق الوحيد هو استدعاء التابع `complete()` قبل تصدير الملف باستخدام التابع `saveToFile()`، وأيضًا اختلاف محتوى الملف المتوقع ضمن المتغير `expectedFileContents` حيث يحوي القيمة `true` بدلًا من `false` عند حقل حالة الاكتمال للمهمة `completed`.

والآن نحفظ الملف ونخرج منه وننفذ الاختبارات بتنفيذ الأمر:

```
npm test
```

سيظهر لنا التالي:

```
...
integrated test
    ✓ should be able to add and complete TODOs

  complete()
    ✓ should fail if there are no TODOs

  saveToFile()
    ✓ should save a single TODO
    ✓ should save a single TODO that's completed


passing (26ms)
```

نجحت الاختبارات كما هو متوقع، لكن يمكننا تحسين طريقة كتابتها وتبسيطها أكثر، حيث نلاحظ أن كلًا من الاختبارين يحتاجان إلى إنشاء كائن من الصنف `Todos` في بداية الاختبار، وفي حال إضافة اختبارات جديدة نحتاج لإنشاء هذا الكائن أيضًا لذا ستتكرر تلك العملية كثيرًا ضمن الاختبارات، وسينتج عن تنفيذ تلك الاختبارات في كل مرة ملفًا جديدًا يتم تصديره وحفظه.

وقد يظن المستخدم لهذه الوحدة خطأً أن هذا الملف هو ملف مهام حقيقية وليس ملف ناتج عن عملية الاختبار، ولحل تلك المشكلة يمكننا حذف الملفات الناتجة بعد انتهاء الاختبار مباشرةً باستخدام الخطافات تلك، حيث نستفيد من الخطاف `beforeEach()` لإعداد المهام قبل اختبارها، وهنا ضمن هذا الخطاف نُعد ونحضر عادةً أي بيانات سنستخدمها داخل الاختبارات، ففي حالتنا نريد إنشاء الكائن `todos` وبداخله مهمة جديدة نجهزها ونضيفها مسبقًا، وسنستفيد من الخطاف `afterEach()` لحذف الملفات الناتجة بعد كل اختبار، لذلك نعدل مجموعة اختبارات التابع `saveToFile()` ضمن ملف الاختبارات index.test.js ليصبح كالتالي:

```
...
describe("saveToFile()", function () {
  beforeEach(function () {
    this.todos = new Todos();
    this.todos.add("save a CSV");
  });

  afterEach(function () {
    if (fs.existsSync("todos.csv")) {
      fs.unlinkSync("todos.csv");
    }
  });

  it("should save a single TODO without error", async function () {
    await this.todos.saveToFile();

    assert.strictEqual(fs.existsSync("todos.csv"), true);
    let expectedFileContents = "Title,Completed\nsave a CSV,false\n";
    let content = fs.readFileSync("todos.csv").toString();
    assert.strictEqual(content, expectedFileContents);
  });

  it("should save a single TODO that's completed", async function () {
    this.todos.complete("save a CSV");
    await this.todos.saveToFile();

    assert.strictEqual(fs.existsSync("todos.csv"), true);
    let expectedFileContents = "Title,Completed\nsave a CSV,true\n";
    let content = fs.readFileSync("todos.csv").toString();
    assert.strictEqual(content, expectedFileContents);
  });
});
```

نلاحظ إضافة الخطاف `beforeEach()` داخل مجموعة الاختبار:

```
...
beforeEach(function () {
```

```
  this.todos = new Todos();
  this.todos.add("save a CSV");
});
...
```

حيث أنشأنا كائنًا جديدًا من الصنف `Todos` سيكون متاحًا لكل الاختبارات ضمن هذه المجموعة، وذلك لأن موكا سيشارك قيمة الكائن `this` الذي أضفنا له خصائص ضمن الخطاف `beforeEach()` مع جميع الاختبارات في توابع الاختبار `it()`، وقيمته ستكون واحدة ضمن مجموعة الاختبارات داخل `describe()`، حيث بالاستفادة من تلك الميزة يمكننا مشاركة بيانات مُعدة مسبقًا مع جميع الاختبارات.

أما داخل الخطاف `afterEach()`، فقد حذفنا ملف CSV الناتج عن الاختبارات:

```
...
afterEach(function () {
    if (fs.existsSync("todos.csv")) {
        fs.unlinkSync("todos.csv");
    }
});
...
```

في حال فشلت الاختبارات فلن يُنشأ ذلك الملف، لهذا السبب نختبر أولًا وجوده باستخدام التابع `existsSync()` قبل تنفيذ عملية الحذف باستخدام التابع `unlinkSync()`، ثم بدلنا في باقي الاختبارات الإشارة إلى كائن المهام `todos` الذي كنا نُنشئه ضمن `it()` مباشرةً، ليشير إلى الكائن الذي أعددناه ضمن الخطاف عن طريق `this.todos`، وحذفنا أسطر إنشاء الكائن `todos` ضمن تلك الاختبارات.

والآن لننفذ تلك الاختبارات بعد التعديلات ونتأكد من نتيجتها بتنفيذ الأمر `npm test` لنحصل على التالي:

```
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs


  saveToFile()
    ✓ should save a single TODO without error
    ✓ should save a single TODO that's completed
```

```
  passing (20ms)
```

نلاحظ أنه لا تغيير في نتائج الاختبار وجميعها نجحت، وأصبحت اختبـارات التـابع `saveToFile()` أبسـط وأسرع بسبب مشاركة الكائن مع جميع الاختبارات، وحللنا مشكلة ملف CSV الناتج عن تنفيذ الاختبارات.

6.6 خاتمة

كتبنا في هذا الفصل وحدة برمجية لإدارة المهام في نود، واختبرنا عملها يدويًا في البداية داخل حلقـة REPL التفاعلية، ثم أنشأنا ملف اختبار واستخدمنا إطار عمل الاختبارات موكا Mocha لأتمتة تنفيـذ جميـع الاختبـارات على تلك الوحدة، واستخدمنا الوحدة `assert` للتوكيد والتحقق من عمل الوحـدة الـتي طورناهـا، وتعلمنـا كيـف يمكن اختبار التوابع المتزامنة واللامتزامنة في موكا Mocha، واستعنا أخيرًا بالخطافات لتبسيط كتابة الاختبارات المرتبطة ببعضها وتسهيل قراءتها وزيادة سهولة التعديل عليها لاحقًا.

والآن حاول عند تطوير برنامجك التالي كتابة الاختبارات لمزاياه، أو يمكنك البدء بكتابة الاختبارات له أولًا من خلال تحديد الدخل والخرج المتوقع من التوابع التي ستكتبها وكتابة اختبار لها على هذا الأساس ثم ابدأ ببنائها.

7. استخدام الوحدة HTTP لإنشاء خادم ويب

يرسل المتصفح عند استعراضك لصفحة ويب ما طلبًا إلى جهاز حاسوب آخر عبر الإنترنت وهو بدوره يرسل الصفحة المطلوبة كجواب لذلك الطلب، حيث ندعو جهاز الحاسوب الذي أُرسل إليه ذلك الطلب بخادم الويب web server، ووظيفته تلقي طلبات HTTP القادمة من العملاء كمتصفحات الويب، ويرسل بالمقابل رد HTTP يحتوي على صفحة HTML أو بيانات بصيغة JSON في حال كان دور الخادم تمثيل واجهة برمجية API، ولإرسال هذه البيانات ومعالجة الطلبات يحتاج خادم الويب لعدة برمجيات تقسم إلى صنفين أساسيين هما شيفرات الواجهات الأمامية Front-end code وهدفها عرض المحتوى المرئي للعميل مثل المحتوى وتنسيق الصفحة من ألوان مستخدمة أو خطوط، والواجهات الخلفية Back-end code وهدفها تحديد طرق تبادل البيانات ومعالجة الطلبات القادمة من المتصفح وتخزينها بالاتصال بقاعدة البيانات، والعديد من العمليات الأخرى.

تتيح لنا بيئة نود Node.js كتابة شيفرات الواجهات الخلفية باستخدام لغة جافاسكربت، والتي كان سابقًا استخدامها محصورًا على تطوير الواجهات الأمامية فقط، وسهل استعمال بيئة نود استخدام لغة جافاسكربت لتطوير الواجهات الأمامية والخلفية معًا عملية تطوير خوادم الويب بدلًا من استعمال لغات أخرى لتطوير الواجهات الخلفية مثل لغة PHP، وهو السبب الأساسي في شهرة نود واستخدامها الواسع لتطوير شيفرات الواجهات الخلفية.

سنتعلم في هذا الفصل كيف نبني خادم ويب بالاستعانة بالوحدة البرمجية http التي توفرها نود يمكنه إعادة صفحات الويب بلغة HTML والبيانات بصيغة JSON وحتى ملفات البيانات بصيغة CSV.

7.1 إنشاء خادم HTTP بسيط في Node.js

سنبدأ بإنشاء خادم ويب يعيد للمستخدم نصًا بسيطًا، لنتعلم بذلك أساسيات إعداد الخادم والتي سنعتمد عليها لتطوير خوادم أخرى ستعيد البيانات بصيغ متقدمة مثل صيغة JSON.

نبدأ بإعداد البيئة البرمجية لتنفيذ التمارين ضمن هذا الفصل فنُنشئ مجلدًا جديدًا بالاسم first-servers ثم ننتقل إليه:

```
mkdir first-servers
cd first-servers
```

ونُنشئ الملف الرئيسي لشيفرة الخادم:

```
touch hello.js
```

نفتح الملف ضمن أي محرر نصوص سنستخدم في هذا الفصل محرر نانو nano:

```
nano hello.js
```

نضيف إلى الملف السطر التالي لاستيراد الوحدة البرمجية `http` التي يوفرها نود افتراضيًا:

```
const http = require("http");
```

تحوي وحدة `http` توابع لإنشاء الخادم سنستخدمها لاحقًا، ويمكنك التعرف أكثر على الوحدات البرمجية بمراجعة الفصل الرابع من هذا الكتاب.

والآن لنعرف ثابتين الأول هو اسم المضيف والثاني هو رقم المنفذ الذي سيستمع إليه الخادم:

```
...
const host = 'localhost';
const port = 8000;
```

كما ذكرنا سابقًا يستقبل الخادم الطلبات المرسلة إليه من متصفح العميل، ويمكن الوصول للخادم عبر عنوانه بإدخال اسم النطاق له والذي سيترجم لاحقًا إلى عنوان IP من قبل خادم DNS، ويتألف هذا العنوان من عدة أرقام متتالية مميزة لكل جهاز ضمن الشبكة مثل شبكة الإنترنت، واسم النطاق `localhost` هو عنوان خاص يشير به جهاز حاسوب إلى نفسه ويقابله عنوان IP التالي `127.0.0.1`، وهو متاح فقط ضمن جهاز الحاسوب المحلي وليس متاحًا على أي شبكة موصول بها بما فيها شبكة الإنترنت.

ويعبر رقم المنفذ port عن بوابة مميزة على الجهاز صاحب عنوان IP المحدد، حيث سنستخدم في حالتنا المنفذ رقم `8000` على الجهاز المحلي لخادم الويب، ويمكن استخدام أي رقم منفذ آخر غير محجوز، ولكن عادة ما نعتمد المنفذ رقم `8080` أو `8000` خلال مرحلة التطوير لخوادم HTTP، وبعد ربط الخادم على اسم المضيف ورقم المنفذ المحددين سنتمكن من الوصول إليه من المتصفح المحلي عبر العنوان http://localhost:8000.

والآن لنضيف دالة مهمتها معالجة طلبات HTTP الواردة وإرسال رد HTTP المناسب لها، حيث تستقبل الدالة معاملين الأول `req` وهو كائن يمثل الطلب الوارد ويحوي البيانات الواردة ضمن طلب HTTP، والثاني

`res` وهو كائن يحوي توابع مفيدة لبناء الرد المراد إرساله للعميل، حيث نستخدمه لإرسال رد HTTP من الخادم، وسنعيد بدايةً الرسالة "My first server!" لكل الطلبات الواردة إلى الخادم:

```
...

const requestListener = function (req, res) {
    res.writeHead(200);
    res.end("My first server!");
};
```

يفضل إعطاء الدوال اسمًا واضحًا يدل على وظيفتها، فمثلًا إذا كان تابع الاستماع للطلب يعيد قائمة من الكتب المتوفرة فيفضل تسميته `listBooks()`، لكن في حالتنا وبما أننا نختبر ونتعلم فيمكننا تسميته بالاسم `requestListener` أي المستمع للطلب.

تستقبل توابع الاستماع للطلبات request listener functions كائنين كمعاملات لها نسميهما عادةً `req` و `res`، حيث يُغلَّف طلب HTTP الوارد من المستخدم ضمن كائن الطلب في أول معامل `req`، ونبني الرد على ذلك الطلب بالاستعانة بكائن الرد في المعامل الثاني `res`.

يعيِّن السطر الأول من تابع الاستماع السابق `res.writeHead(200);` رمز الحالة لرد HTTP الذي سنرسله، والذي يحدد حالة معالجة الطلب من قبل الخادم، ففي حالتنا وبما أن الطلب سينجح ويكون صحيح دومًا نعين للرد رمز الحالة `200` والذي يعني إتمام الطلب بنجاح أو `"OK"`، وانظر مقال رموز الإجابة في HTTP للتعرف على أهم رموز الإجابة في طلبات HTTP.

أما السطر الثاني من التابع `res.end("My first server!");` فيرسل الرد للعميل الذي أرسل الطلب، ويمكن باستخدام ذلك التابع إرسال البيانات التي يجب أن يرسلها الخادم ضمن الرد وفي حالتنا هي إرسال نص بسيط.

والآن أصبحنا جاهزين لإنشاء الخادم والاستفادة من تابع الاستماع السابق:

```
...

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

نحفظ الملف ونخرج منه، وفي حال كنت تستخدم محرر النصوص `nano` يمكنك الخروج بالضغط على الاختصار CTRL+X.

في الشيفرة السابقة، أنشأنا في أول سطر كائن الخادم `server` باستخدام التابع `createServer()` من الوحدة `http`، وظيفته استقبال طلبات HTTP وتمريرها إلى تابع الاستماع `requestListener()`، وبعدها نربط الخادم إلى عنوان الشبكة الذي سيستمع إليه باستخدام التابع `server.listen()` ويمكن أن نمرر له رقم المنفذ `port` كمعامل أول، وعنوان الشبكة `host` كمعامل ثانٍ، وفي النهاية دالة رد نداء callback تُستدعى عند بدء الاستماع من قبل الخادم للطلبات الواردة، وكل تلك المعاملات اختيارية لكن يفضل تمريرها وتحديد قيمها ليتضح عند قراءة الشيفرة على أي منفذ وعنوان سيستمع الخادم، ومن الضروري معرفة هذه الإعدادات للخادم عند نشر خادم الويب في بعض البيئات، خاصة التي تحتاج لإعداد موزع الحمل load balancing وإعداد الأسماء في خدمة DNS، ومهمة دالة رد النداء التي مررناها هناك طباعة رسالة إلى الطرفية تبين أن الخادم بدأ الاستماع مع توضيح عنوان الوصول إليه.

يجب الملاحظة أنه حتى ولو لم نكن بحاجة لاستخدام كائن الطلب `req` ضمن تابع الاستماع، فمن الضروري تمريره كمعامل أول حتى نتمكن من الوصول لكائن الرد `res` كمعامل ثانٍ بشكل صحيح.

رأينا مما سبق سهولة إنشاء خادم ويب في نود حيث استطعنا بأقل من 15 سطرًا تجهيز خادم الويب، والآن لنشغله ونرى كيف يعمل بتنفيذ الأمر التالي:

```
node hello.js
```

سيظهر لنا الخرج التالي ضمن الطرفية:

```
Server is running on http://localhost:8000
```

نلاحظ أن سطر الأوامر خرج من وضع الإدخال الافتراضي، لأن خادم الويب يعمل ضمن إجرائية طويلة لا تنتهي ليتمكن من الاستماع إلى الطلبات الواردة إليه في أي وقت، أما عند حدوث خطأ ما أو في حال أوقفنا الخادم يدويًا سيتم بذلك الخروج من تلك الإجرائية، لهذا السبب يجب اختبار الخادم من طرفية أخرى جديدة عبر التواصل معه باستخدام أداة تتيح إرسال واستقبال البيانات عبر الشبكة مثل cURL، وباستخدامها ننفذ الأمر التالي لإرسال طلب HTTP من نوع GET لخادم الويب السابق:

```
curl http://localhost:8000
```

بعد تنفيذ الأمر سيظهر لنا رد الخادم ضمن الخرج كالتالي:

```
My first server!
```

نلاحظ ظهور الرد من طرف الخادم، ونكون بذلك قد أعددنا خادم الويب واختبرنا إرسال طلب إليه واستقبال الرد منه بنجاح، لكن لنفصّل أكثر في تلك عملية ونفهم ما حدث.

عنـد إرسـال طلب الاختبـار إلى الخـادم أرسـلت الأداة cURL طلب HTTP من النـوع GET إلى الخـادم على العنوان http://localhost:8000، ثم استقبل خادم الويب الذي أنشأناه ذلك الطلب من العنـوان الـذي يسـتمع عليه ومرره إلى تابع الاستماع ومعالجة الطلبات المحـدد `requestListener()`، وهـو بـدوره عيّن رمـز الحالـة بالرقم 200 وأرسل البيانات النصية ضمن الرد، ثم أرسل الخادم بعدها الرد إلى صاحب الطلب وهو الأداة cURL، والتي بدورها عرضت محتواه على الطرفية.

نوقف الخادم الآن بالضغط على الاختصار CTRL+C ضمن الطرفية الخاصة به لإيقاف الإجرائيـة الـتي يعمـل ضمنها ونعود بذلك إلى سطر الأوامر بحالته الافتراضية لاستقبال كتابة الأوامر وتنفيذها، ولكن ما طورناه يختلـف عن خوادم الويب للمواقع التي نزورها عادة أو الواجهات البرمجيـة API الـتي نتعامـل معهـا، فهي لا ترسـل نصًـا بسيطًا فحسب بل إما ترسل لنا صفحات مكتوبة بلغة HTML أو بيانات بصيغة JSON، لذلك في سنتعلم الفقرة التالية كيف يمكننا الرد ببيانات مكتوبة بتلك الصيغ الشائع استخدامها على شبكة الويب.

7.2 الرد بعدة أنواع من البيانات

يمكن لخادم الويب إرسال البيانات للعميل ضمن الرد بعدة صيغ منها HTML و JSON وحتى XML وصـيغة CSV، كما يمكن للخوادم إرسال بيانات غير نصية مثل مستندات PDF أو الملفـات المضـغوطة وحـتى الصـوت أو الفيديو.

سنتعلم في هذه الفقرة كيف نرسل بعض الأنواع من تلـك البيانـات وهي JSON و CSV وصـفحات HTML وهي صيغ البيانات النصية الشائع استخدامها في الويب، حيث توفر العديد من الأدوات ولغـات البرمجـة دعمًـا واسعًا لإرسال تلك الأنواع من البيانات ضمن ردود HTTP، فمثلًا يمكن إرسالها في نود باتباع الخطوات التالية:

1. تعيين قيمة لترويسة `Content-Type` للرد في HTTP بقيمة تناسب نوع المحتوى المُرسل.
2. تمرير البيانات بالصيغة الصحيحة للتابع `res.end()` لإرسالها.

سنطبق ذلك في عدة أمثلة لاحقة، حيث ستتشارك كل تلك الأمثلة في نفس طريقة إعداد الخادم كما فعلنا في الفقرة السابقة، والاختلاف بينها سيكون ضمن تـابع معالجـة الطلب فقـط `requestListener()`، لـذلك سنحضر ملفات تلك الأمثلة باسـتخدام قـالب موحـد لهـا جميعًـا سـنكتبه في البدايـة، لهـذا نبـدأ بإنشـاء ملـف جافاسكربت جديد بالاسم html.js سيحتوي على مثال إرسال الخادم لبيانات بصيغة HTML.

نبدأ بكتابة الشيفرات المشتركة بين جميع الأمثلـة ضـمنه ثم ننسـخ الملـف إلى عـدة نسـخ لتجهـيز ملفـات الأمثلة الباقية:

```
touch html.js
```

نفتح الملف ضمن أي محرر نصوص:

```
nano html.js
```

ونضع داخله محتوى القالب لجميع الأمثلة اللاحقة كالتالي:

```
const http = require("http");

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

نحفظ الملف ونخرج منه وننسخه إلى ملفين جديدين الأول لمثال إرسال البيانات بصيغة CSV ضمن الرد كالتالي:

```
cp html.js csv.js
```

والآخر لإرسال البيانات بصيغة JSON:

```
cp html.js json.js
```

ونحضر الملفات التالية أيضًا والتي سنستخدمها للأمثلة في الفقرة اللاحقة:

```
cp html.js htmlFile.js
cp html.js routes.js
```

بذلك نكون قد جهزنا جميع ملفات الأمثلة وبإمكاننا البدء بتضمينها، وسنبدأ في أول مثال بالتعرف على طريقة إرسال البيانات بصيغة JSON.

7.2.1 إرسال البيانات بصيغة JSON

صيغة ترميز كائنات جافاسكربت objects أو ما يعرف بصيغة JSON هي صيغة نصية لتبادل البيانات، وكما يشير اسمها فهي مشتقة من كائنات جافاسكربت ولكن يمكن التعامل معها من أي لغة برمجة أخرى تدعمها وقادرة على تحليل صيغتها، وهي تستخدم عادة في عمليات إرسال واستقبال البيانات من الواجهات

البرمجية للتطبيقات API، ومن أسباب انتشارها صغر حجم البيانات عند إرسالها بهذه الصيغة مقارنة بالصيغ الأخرى مثل XML مثلًا، ومما يساعد في التعامل معها بكل سهولة هو توفر الأدوات لقراءة وتحليل هذه الصيغة.

والآن نفتح ملف المثال json.js:

```
nano json.js
```

وبما أننا نريد إرسال البيانات بصيغة JSON لنعدل تابع معالجة الطلب `requestListener()` ليعين قيمة الترويسة المناسبة لردود JSON كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
};
...
```

يضيف التابع `res.setHeader()` ترويسة HTTP إلى الرد توفر معلومات إضافية عن الطلب أو الرد المرسل، حيث يمرر له معاملين هما اسم الترويسة وقيمتها، حيث تصف قيمة الترويسة `Content-Type` صيغة البيانات أو نوع الوسائط media type المرفقة ضمن جسم الطلب، وفي حالتنا يجب تعيين قيمة الترويسة إلى `application/json`، ثم نعيد بعدها البيانات بصيغة JSON إلى المستخدم كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    res.writeHead(200);
    res.end(`{"message": "This is a JSON response"}`);
};
...
```

ضبطنا كما المثال السابق رمز الرد إلى القيمة `200` للدلالة على نجاح العملية، والفرق هنا أننا مررنا لتابع إرسال البيانات ضمن الرد `response.end()` سلسلة نصية تحوي بيانات بصيغة JSON.

والآن نحفظ الملف ونخرج منه ونشغل الخادم بتنفيذ الأمر التالي:

```
node json.js
```

ونفتح طرفية أخرى لتجربة إرسال طلب إلى الخادم باستخدام الأداة cURL كالتالي:

```
curl http://localhost:8000
```

بعد إرسال الطلب والضغط على زر الإدخال ENTER نحصل على النتيجة التالية:

```
{"message": "This is a JSON response"}
```

نكون بذلك قد تعلمنا كيف يمكن إرسال رد يحوي بيانات بصيغة JSON مثل ما تفعل الواجهات البرمجية للتطبيقات API تمامًا.

وبعد الاختبار نوقف الخادم بالضغط على الاختصار CTRL+C لنعود إلى سطر الأوامر مجددًا، حيث سنتعلم في الفقرة التالية كيف يمكن إرسال البيانات بصيغة CSV هذه المرة.

7.2.2 إرسال البيانات بصيغة CSV

شاع استخدام صيغة القيم المفصولة بفاصلة أو CSV عند التعامل مع البيانات المجدولة ضمن جداول، حيث يُفصل بين السجلات ضمن الجدول سطر جديد، وبين القيم على نفس السطر بفاصلة.

والآن نفتح ملف المثال csv.js ضمن محرر النصوص ونعدل طريقة إرسال الطلب ضمن التابع `requestListener()` كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/csv");
    res.setHeader("Content-Disposition",
"attachment;filename=oceanpals.csv");
};
...
```

نلاحظ كيف حددنا قيمة الترويسة `Content-Type` هذه المرة بالقيمة `text/csv` والتي تدل على أن البيانات المرسلة مكتوبة بصيغة CSV، وأضفنا هذه المرة ترويسة جديدة بالاسم `Content-Disposition` لتدل المتصفح على طريقة عرض البيانات المرسلة إليه، فإما أن تبقى ضمن المتصفح نفسه أو يتم حفظها في ملف خارجي، وحتى لو لم نعين قيمة للترويسة `Content-Disposition` فمعظم المتصفحات الحديثة ستُنزِّل البيانات وتحفظها ضمن ملف تلقائيًا في حال كانت بصيغة CSV، ويسمح تعيين قيمة لهذه الترويسة بتحديد اسم للملف الذي سيتم حفظه، والقيمة التي عيناها تخبر المتصفح أن البيانات المرسلة هي ملف مرفق بصيغة CSV يجب تنزيله وحفظه بالاسم oceanpals.csv.

والآن لنرسل بيانات CSV ضمن الرد كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/csv");
```

```
    res.setHeader("Content-Disposition",
"attachment;filename=oceanpals.csv");
    res.writeHead(200);
    res.end(`id,name,email\n1,Hassan Shark,shark@ocean.com`);
};
...
```

حددنا كما العادة رمز الحالة 200 ضمن الرد للدلالة على نجاح العملية، ومررنا سلسلة نصية تحوي على بيانات بصيغة CSV إلى تابع إرسال البيانات `res.end()`، ونلاحظ كيف يفصل بين تلك القيم فواصل، وبين أسطر الجدول محرف `\n` الذي يدل على سطر جديد، والبيانات التي أرسلناها تحوي سطران الأول فيه ترويسات الجدول والثاني يحوي البيانات الموافقة لها.

والآن لنختبر عمل الخادم لذا نحفظ الملف ونخرج منه وننفذ أمر تشغيل الخادم كالتالي:

```
node csv.js
```

ونفتح طرفية أخرى لتجربة إرسال طلب إلى الخادم باستخدام الأداة cURL كالتالي:

```
curl http://localhost:8000
```

يظهر لنا الرد التالي:

```
id,name,email
1,Hassan Shark,shark@ocean.com
```

إذا حاولنا الوصول للخادم من المتصفح عن طريق العنوان `http://localhost:8000` نلاحظ كيف سيتم تنزيل ملف CSV المرسل وسيحدد تلقائيًا الاسم oceanpals.csv له.

نوقف الخادم الآن لنعود إلى سطر الأوامر مجددًا.

والآن بعد أن تعرفنا على طريقة إرسال البيانات بالصيغ JSON و CSV وهي أشيع الصيغ المستخدمة عند تطوير الواجهات البرمجية API، سنتعرف في الفقرة التالية على طريقة إرسال البيانات بحيث يمكن للمستخدم استعراضها ضمن المتصفح مباشرة.

7.2.3 إرسال البيانات بصيغة HTML

تعد لغة ترميز النصوص الفائقة HTML صيغة لترميز صفحات الويب والتي تتيح للمستخدم التفاعل مع الخادم مباشرةً من داخل المتصفح، ووظيفتها توصيف بنية محتوى الويب حيث تعتمد المتصفحات في عرضها لصفحات الويب على لغة HTML وعلى تنسيقها باستخدام CSS وهي تقنية أخرى من تقنيات الويب وظيفتها تجميل الصفحات وضبط طريقة عرضها.

والآن نفتح ملف المثال لهذه الفقرة html.js ضمن محرر النصوص ونعدل طريقة إرسال الرد ضمن التابع `requestListener()` بداية بتعيين قيمة مناسبة للترويسة `Content-Type` لتدل على صيغة HTML كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
};
...
```

ونعيد بعدها البيانات بصيغة HTML إلى المستخدم بإضافة التالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(`This is HTML`);
};
...
```

كما العادة ضبطنا بداية رمز الحالة لرد HTTP، ثم أرسلنا بيانات بصيغة HTML بتمريرها كسلسلة نصية للتابع `response.end()`، وإذا اختبرنا الاتصال بالخادم عبر المتصفح ستظهر لنا صفحة HTML تحتوي على ترويسة بالنص "This is HTML".

والآن نحفظ الملف ونخرج منه ونشغل الخادم لاختبار ذلك بتنفيذ الأمر التالي:

```
node html.js
```

نطلب بعد تشغيل الخادم عنوانه من المتصفح http://localhost:8000 لتظهر لنا الصفحة التالية:

This is HTML

نوقف الخادم لنعود إلى سطر الأوامر مجددًا، وبذلك نكون تعلمنا طريقة إرسال صفحة HTML عبر كتابة محتواها يدويًا ضمن سلسلة نصية، ولكن عادة نخزن محتوى تلك الصفحات ضمن ملفات HTML منفصلة عن شيفرة الخادم، لذا سنتعرف في الفقرة التالية على طريقة تنفيذ ذلك.

7.3 إرسال ملف صفحة HTML

يمكن إرسال محتوى صفحات HTML عبر تمريرها مباشرة على شكل سلسلة نصية لتابع الإرسال كما فعلنا في الفقرة السابقة، ولكن يفضل تخزين محتوى صفحات HTML ضمن ملفات منفصلة وتخديم محتواها من قبل الخادم، حيث يمكن بذلك التعديل على محتواها بسهولة أكبر، ونكون قد فصلنا بذلك محتوى صفحات الويب عن شيفرات الخادم، وعملية الفصل هذه شائعة في معظم أطر العمل المشهورة لذا سيفيدنا معرفة الطريقة التي يتم بها تحميل وإرسال ملفات HTML.

ولتخديم ملفات HTML من الخادم، يجب تحميل ملفاتها أولًا باستخدام الوحدة fs وكتابة محتوى الملف ضمن رد HTTP، لذا نُنشئ بداية ملف HTML الذي سيرسله الخادم كالتالي:

```
touch index.html
```

نفتح ملف الصفحة index.html ضمن محرر النصوص ونكتب صفحة HTML بسيطة تحتوي على خلفية باللون البرتقالي وعبارة ترحيب في المنتصف كالتالي:

```
<!DOCTYPE html>

<head>
    <title>My Website</title>
    <style>
        *,
        html {
            margin: 0;
            padding: 0;
            border: 0;
        }

        html {
            width: 100%;
            height: 100%;
        }

        body {
            width: 100%;
            height: 100%;
            position: relative;
```

```
            background-color: rgb(236, 152, 42);
        }

        .center {
            width: 100%;
            height: 50%;
            margin: 0;
            position: absolute;
            top: 50%;
            left: 50%;
            transform: translate(-50%, -50%);
            color: white;
            font-family: "Trebuchet MS", Helvetica, sans-serif;
            text-align: center;
        }

        h1 {
            font-size: 144px;
        }

        p {
            font-size: 64px;
        }
    </style>
</head>

<body>
    <div class="center">
        <h1>Hello Again!</h1>
        <p>This is served from a file</p>
    </div>
</body>

</html>
```

ستعرض الصفحة السابقة سطران هما "Hello Again!" و "This is served from a file"، في منتصف الصفحة فوق بعضهما بعضًا، والسطر الأول منها سيظهر بحجم خط أكبر من السطر الآخر، وستظهر النصوص باللون الأبيض وخلفية الصفحة باللون البرتقالي.

والآن نحفظ الملف ونخرج منه ونعود إلى شيفرة الخادم حيث في هذا المثال سنستخدم الملف htmlFile.js الذي أعددناه سابقًا لتطوير الخادم، لذا نفتحه ضمن محرر النصوص ونبدأ أولًا باستيراد الوحدة `fs` بما أننا ننوي قراءة الملف السابق:

```
const http = require("http");
const fs = require('fs').promises;
...
```

سنستفيد من التابع `readFile()` لتحميل محتوى ملف HTML، ونلاحظ كيف استوردنا نسخة التوابع التي تستعمل الوعود وذلك لتبسيط كتابة الشيفرات، حيث أنها أسهل بالقراءة من استخدام توابع رد النداء، والتي سيتم استيرادها افتراضيًا في حال استوردنا الوحدة `fs` فقط كالتالي `require('fs')`، ويمكنك الرجوع إلى الفصل الخامس من هذا الكتاب للتعرف أكثر على البرمجة اللامتزامنة في جافاسكربت.

والآن نبدأ بقراءة ملف HTML السابق عند وصول طلب من المستخدم، لهذا نعدل تابع معالجة الطلب `requestListener()` كالتالي:

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
};
...
```

استدعينا التابع `fs.readFile()` لتحميل الملف، ومررنا له القيمة `__dirname + "/index.html"` والتي يدل فيها المتغير الخاص `__dirname` على المسار المطلق للمجلد الحاوي على ملف جافاسكربت الحالي، ونضيف إليه القيمة `/index.html` للحصول على المسار المطلق الكامل لملف HTML الذي نريد إرساله، وبعد اكتمال تحميل الملف نضيف التالي:

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
```

```
            res.end(contents);
        })
};
...
```

سنرسل المحتوى الناتج عن نجاح تنفيذ الوعد الذي يعيده التابع `fs.readFile()` أي قراءة الملف بنجاح كما فعلنا سابقًا وذلك ضمن التابع `then()`، حيث سيحتوي العامل `contents` على بيانات الملف بعد نجاح قراءته.

وكما فعلنا سابقًا ضبطنا بدايةً قيمة الترويسة `Content-Type` إلى `text/html` للدلالة على إرسال محتوى بصيغة HTML، ثم ضبطنا رمز الحالة إلى `200` للدلالة على نجاح الطلب، ثم أرسلنا صفحة HTML التي حملناها إلى المستخدم وتحديدًا محتوى المتغير `contents`، لكن أحيانًا قد يفشل التابع `fs.readFile()` في قراءة الملف لأي سبب كان، لذا يجب معالجة حالة الخطأ تلك بإضافة الشيفرة التالية ضمن التابع `requestListener()`:

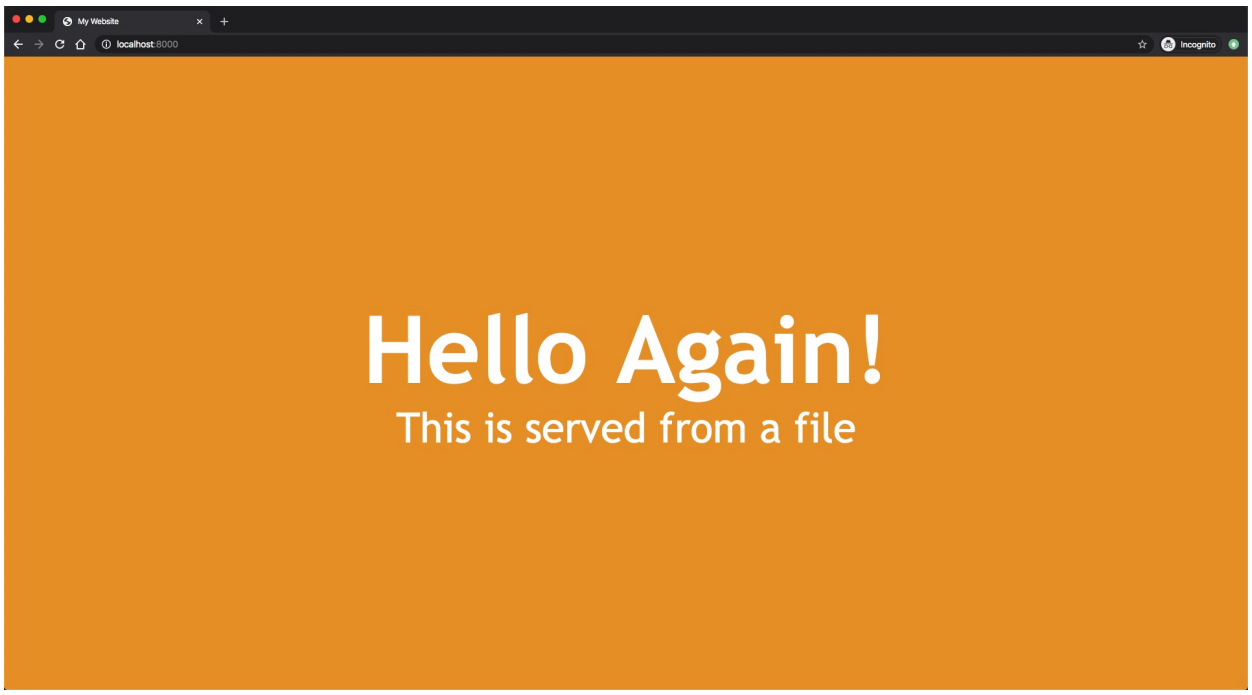
```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(contents);
        })
        .catch(err => {
            res.writeHead(500);
            res.end(err);
            return;
        });
};
...
```

نحفظ الملف ونخرج من محرر النصوص، ونلاحظ عندما يحدث خطأ ما أثناء تنفيذ الوعد سيتم رفضه، حيث يمكننا معالجة الخطأ باستخدام التابع `catch()` والذي يُمرر إليه كائن الخطأ الذي يرميه استدعاء تابع قراءة الملف `fs.readFile()`، ونحدِّد فيه رمز حالة الرد بالقيمة `500` للدلالة على حدوث خطأ داخلي من طرف الخادم ونعيد الخطأ للمستخدم.

والآن نشغل الخادم كالتالي:

```
node htmlFile.js
```

ونزور عنوانه http://localhost:8000 باستخدام المتصفح ستظهر لنا صفحة الويب كالتالي:



وبذلك نكون قد أرسلنا صفحة HTML مُخزَّنة من ملف إلى المستخدم، والآن نوقف الخادم ونعود إلى الطرفية مجددًا.

انتبه إلى أنَّ تحميل صفحة HTML بهذه الطريقة عند كل طلب HTTP يصل إلى الخادم يؤثر على الأداء، ومع أن الصفحة التي استخدمناها في مثالنا حجمها صغير وهو حوالي 800 بايت فقط، إلا أنه عند بناء التطبيقات قد يصل أحيانًا حجم الصفحات المستخدمة إلى رتبة الميجابايت، مما يؤدي لبطء في تحميلها وتخديمها للعميل، خصوصًا إذا كان من المتوقع ورود طلبات كثيرة إلى الخادم، لذا ولرفع الأداء يمكن تحميل محتوى الملفات مرة واحدة عند إقلاع الخادم وإرسال محتواها للطلبات الواردة، وبعد انتهاء عملية التحميل نخبر الخادم ببدء الاستماع للطلبات على العنوان المحدد له، وهذا ما سنتعلمه في الفقرة التالية حيث سنطور هذه الميزة في الخادم لرفع أداءه.

7.3.1 رفع كفاءة تخديم صفحات HTML

بدلًا من تحميل ملفات HTML عند كل طلب يرد إلى الخادم يمكننا تحميلها لمرة واحدة فقط في البداية، وبعدها نعيد تلك البيانات المخزنة مسبقًا لكل طلب سيرد لاحقًا إلى الخادم، لذلك نعود لملف المثال السابق htmlFile.js ونفتحه ضمن محرر النصوص ونضيف فيه متغيرًا جديدًا قبل إنشاء تابع معالجة الطلب `requestListener()`:

```
...
let indexFile;

const requestListener = function (req, res) {
...
```

سيحتوي هذا المتغير على محتويات ملف HTML عند تشغيل الخادم، والآن نعدل على التابع `requestListener()` وبدلًا من تحميل الملف داخله نعيد مباشرة محتوى المتغير `indexFile`:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(indexFile);
};
...
```

ونبدل مكان شيفرة تحميل الملف من داخل التابع `requestListener()` إلى أعلى الملف في مكان إعداد الخادم ليصبح كالتالي:

```
...

const server = http.createServer(requestListener);

fs.readFile(__dirname + "/index.html")
    .then(contents => {
        indexFile = contents;
        server.listen(port, host, () => {
            console.log(`Server is running on http://${host}:$
{port}`);
        });
    })
    .catch(err => {
        console.error(`Could not read index.html file: ${err}`);
        process.exit(1);
    });
```

نلاحظ أن عملية قراءة الملف شبيهة جدًا بما نفذنا سابقًا، ولكن الفرق هنا أننا نحفظ بعد نجاح عمليـة قـراءة الملف محتوياته ضمن المتغير العام `indexFile`، وبعد ذلك نشغل الخادم باستدعاء التابع `listen()`، حيث أن الخطـوة الأساسـية هي تحميـل الملـف لمـرة واحـدة قبـل تشـغيل الخـادم، لنضـمن بـذلك أن التـابع `requestListener()` سيعيد محتوى الملف المخزن ضمن المتغير `indexFile` وأن قيمته ليست فارغة.

وعدلنا أيضًا تابع معالجة الخطأ بحيث عند حدوث أي خطأ في عملية تحميل الملـف سـنطبع رسـالة ضـمن الطرفية توضح السبب ونخرج مباشرة من الخادم عبر استدعاء التـابع `exit()`، وبـذلك نسـتطيع معاينـة سـبب الخطأ الذي يمنع تحميل الملف ونعالج المشـكلة أولًا ثم نعيـد تشـغيل الخـادم بنجـاح، فمـا الفائـدة من تشـغيل الخادم دون تحميل الملف المراد تخديمه.

أنشأنا في الأمثلة السابقة عدة خوادم ويب تعيد كل منها المحتوى بصيغة مختلفة للمستخدم، ولم نستخدم حتى الآن أي بيانات من الطلب القادم إلى الخادم لتحديد ما يطلبه المستخدم تمامًا، حيث تفيدنا تلـك البيانـات في عملية التوجيه وإعداد عدة مسـارات يسـتطيع خـادم الـويب الواحـد تخـديمها وهـذا تمامًـا مـا سـنتعلمه في الفقرة التالية.

7.4 إدارة الوجهات Routes في الخادم

معظم المواقع التي نزورها أو الواجهات البرمجية التي نتعامل معها تحوي عدة مسارات أو وجهات تسمح لنا بالوصول إلى عدد من الموارد على نفس الخادم، فمثلًا في نظام لإدارة الكتب في المكتبات على النظـام أن يـدير بيانات الكتب وبيانات أخرى مثل المؤلفين لهذه الكتب، وسيوفر خدمات أخرى مثل البحث والتصنيف، ومع أن بيانات الكتب والمؤلفين لها مرتبطة ببعضها لكن يمكن معاملتها كمَوردين مختلفين، وفي هذه الحالـة يمكن أن نطور النظام ليخدّم كل نوع من تلك الموارد ضمن مسار محدد له، ليمـيز المسـتخدم الـذي يتعامـل مـع الواجهـة البرمجية API للنظام نوع المورد الذي يتعامل معه.

لنطبق المثال ذاك ببناء خـادم بسـيط لنظـام إدارة مكتبـة سـيحتوي على نـوعين من البيانـات، فعنـد طلب المستخدم المورد من المسار `/books` سنرسل له قائمة بالكتب المتوفرة بصيغة JSON، أما عند طلب المسـار `/authors` سنرسل له قائمة بمعلومات حـول المـؤلفين بصـيغة JSON أيضًـا، ففي كـل أمثلـة خـوادم الـويب السابقة في هذا الفصل كنا نرسل نفس الرد دومًا لكل الطلبات التي تصل إلى الخادم.

لنختبر ذلك، علينا أولًا إرسال طلبات مختلفة للخادم ونلاحظ الرد المرسل على كـل منهـا، لـذا نعيـد تشـغيل خادم JSON الذي طورناه سابقًا بتنفيذ الأمر:

```
node json.js
```

وكالعادة في طرفية أخرى نرسل طلب HTTP باستخدام cURL كالتالي:

```
curl http://localhost:8000
```

يعيد لنا الخادم الرد التالي:

```
{"message": "This is a JSON response"}
```

لنختبر الآن إرسال طلب على مسار مختلف للخادم كالتالي:

```
curl http://localhost:8000/todos
```

سنلاحظ ظهور نفس الرد السابق:

```
{"message": "This is a JSON response"}
```

ذلك لأن الخادم لا يعير اهتمامًا أبدًا عند معالجة الطلب داخل التابع `requestListener()` للمسار الذي يطلبه المستخدم ضمن URL، لذا عندما أرسلنا طلبًا إلى المسار `/todos` أعاد لنا الخادم نفس محتوى JSON الذي يعيده افتراضيًا، ولكن لبناء خادم نظام إدارة المكتبة يجب أن نفصل ونحدد نوع البيانات التي سنعيدها للمستخدم بناءً على المسار الذي يطلب الوصول إليه.

والآن نوقف الخادم ونفتح الملف routes.js ونبدأ بتخزين بيانات JSON التي سيوفرها الخادم ضمن متغيرات قبل تعريف تابع معالجة الطلب `requestListener()` كالتالي:

```
...
const books = JSON.stringify([
    { title: "The Alchemist", author: "Paulo Coelho", year: 1988 },
    { title: "The Prophet", author: "Kahlil Gibran", year: 1923 }
]);

const authors = JSON.stringify([
    { name: "Paulo Coelho", countryOfBirth: "Brazil", yearOfBirth:
1947 },
    { name: "Kahlil Gibran", countryOfBirth: "Lebanon", yearOfBirth:
1883 }
]);
...
```

يحتوي المتغير `books` على سلسلة نصية بصيغة JSON فيها مصفوفة من الكائنات التي تمثل الكتب المتوفرة، ويحتوي كل كتاب منها على خاصية العنوان أو الاسم والمؤلف وسنة النشر، بينما يحتوي المتغير `authors` على سلسلة نصية بصيغة JSON أيضًا فيها مصفوفة من الكائنات التي تمثل المؤلفين ويملك كل مؤلف منها خاصية اسمه وبلد وسنة الولادة.

وبعد أن جهزنا البيانات التي سنعيدها للمستخدم نبدأ بتعديل تابع معالجة الطلب `requestListener()` ليعيد البيانات المناسبة منها بحسب المسار المطلوب، لذا نبدأ بتعيين قيمة الترويسة `Content-Type` لكل الطلبات التي سنرسلها، وبما أن جميع البيانات هي بصيغة JSON يمكننا تحديد قيمة الترويسة مباشرةً في البداية كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
}
...
```

والآن سنعيد بيانات JSON بحسب المسار المقابل ضمن عنوان URL الذي يحاول المستخدم طلبه، لذا نكتب تعليمة تبديل `switch` بحسب عنوان URL للطلب كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {}
}
...
```

نلاحظ كيف يمكننا الوصول للمسار الذي يطلبه المستخدم من الخاصية `url` من كائن الطلب `req`، ونضيف بعدها حالات التوجيه للمسارات أو الوجهات المحددة ضمن تعليمة `switch` ونعيد بيانات JSON المناسبة لها، حيث توفر التعليمة `switch` في جافاسكربت طريقة للتحكم بالشيفرات التي ستنفَّذ بحسب القيمة أو التعبير البرمجي الممرر لها بين القوسين.

والآن نضيف الحالة التي يطلب بها المستخدم قائمة الكتب باستخدام الكلمة `case` كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
    }
```

```
}
...
```

نعين عندها رمز الحالة للطلب بالقيمة 200 للدلالة على نجاح الطلب ونعيد قيمة JSON الحاوية على قائمة الكتب المتاحة، ونضيف بعدها حالة case أخرى للرد على مسار طلب المؤلفين كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
        case "/authors":
            res.writeHead(200);
            res.end(authors);
            break
    }
}
...
```

كما في الحالة السابقة نضبط أولًا رمز الحالة للرد بالقيمة 200 للدلالة على صحة الطلب، ونعيد قيمة JSON الحاوية على قائمة المؤلفين، وفي حال طلب المستخدم أي مسار آخر غير مدعوم سنرسل له خطأ، ولهذه الحالة يمكن إضافة الحالة الافتراضية default لالتقاط كل الحالات التي لا تطابق أي من الحالات المُعرّفة حيث نضبط فيها رمز الحالة إلى القيمة 404 للدلالة على أن المورد الذي يحاول المستخدم الوصول إليه غير موجود ونعيد رسالة خطأ للمستخدم ضمن كائن بصيغة JSON السابقة كالتالي:

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
        case "/authors":
```

```
            res.writeHead(200);
            res.end(authors);
            break
        default:
            res.writeHead(404);
            res.end(JSON.stringify({error:"Resource not found"}));
    }
}
...
```

والآن لنشغل الخادم ونختبره من طرفية أخرى بإرسال طلب وصول إلى مسار الكتب المتاحة ونعاين الرد:

```
curl http://localhost:8000/books
```

لنحصل على الخرج:

```
[{"title":"The Alchemist","author":"Paulo Coelho","year":1988},
{"title":"The Prophet","author":"Kahlil Gibran","year":1923}]
```

حصلنا على قائمة الكتب كما هو متوقع، وبالمثل نختبر مسار طلب المؤلفين `/authors` كالتالي:

```
curl http://localhost:8000/authors
```

لنحصل على الخرج التالي:

```
[{"name":"Paulo Coelho","countryOfBirth":"Brazil","yearOfBirth":1947},
{"name":"Kahlil
Gibran","countryOfBirth":"Lebanon","yearOfBirth":1883}]
```

وأخيرًا نختبر الوصول إلى مسار غير مدعوم ونتأكد من أن تابع معالجة الطلب `requestListener()` سيعيد لنا رسالة خطأ:

```
curl http://localhost:8000/notreal
```

سيعيد لنا الخادم رسالة الخطأ كالتالي:

```
{"error":"Resource not found"}
```

نوقف الخادم وبذلك نكون قد طورنا خادمًا يمكنه توجيه الطلب ضمن عدة مسارات مدعومة والرد عليها ببيانات مختلفة، وأضفنا إليه أيضًا ميزة إرسال رسالة خطأ عندما يحاول المستخدم الوصول لمسار غير مدعوم.

7.5 خاتمة

طورنا في هذا الفصل عددًا من خوادم HTTP في بيئة نود، حيث بدأنا بإعادة نص بسيط ضمن الرد مرورًا بعدة أنواع من صيغ البيانات مثل JSON و CSV وصفحات HTML، وطورنا الخادم لتحميل صفحات HTML من ملفات خارجية مخصصة لها وتخديمها وإرسال محتواها إلى العميل، وأخيرًا طورنا واجهة برمجية API يمكنها الرد على طلب المستخدم بعدة أنواع من المعلومات بحسب معلومات من الطلب المُرسل للخادم.

وبذلك تكون قد تعلمت طريقة إنشاء خوادم ويب يمكنها معالجة عدة أنواع من الطلبات والردود، والآن حاول مما تعلمت بناء خادم ويب يُخدّم عدة صفحات HTML للمستخدم بحسب المسارات المختلفة التي يطلبها، ويمكنك أيضًا بناء واجهة برمجة التطبيقات API الخاصة بك، ويمكنك الرجوع إلى التوثيق الرسمي للوحدة http من نود لتعلم المزيد عن خوادم الويب.

8. استخدام المخازن المؤقتة Buffers

يمثل المخزن المؤقت buffer مساحة ما في الذاكرة RAM تحتوي على البيانات بالصيغة الثنائيــة binary، ويمكن لنود Node.js أن تتعامل مع هذه الذاكرة باستخدام الصنف Buffer، حيث يمثل البيانـات كسلسـلة من الأعداد بطريقة مشابهة لعمل المصفوفات في جافاسكربت، إلا أن الفرق أن هذه البيانات لا يمكن التعـديل على حجمها بعد إنشاء المخزن، وكثيرًا ما نتعامل مع المخـازن المؤقتـة عنـد تطـوير الـبرامج ضـمن بيئـة نـود دون أن نشعر، فمثلًا عند قراءة ملف ما باستخدام التابع fs.readFile() فسيمرر كائن من نوع مخزن مؤقت يحوي بيانات الملف الذي نحاول قراءته إلى تابع رد النـداء callback أو كنتيجـة للوعـد Promise، وحـتى عنـد إنشـاء طلبـات HTTP فالنتيجة هي مجرى stream من البيانات المخزنة مؤقتًا في مخزن مؤقت داخلي يساعد المستخدم على معالجة بيانات جواب الطلب على دفعات بدلًا من دفعة واحدة.

ونستفيد من المخازن المؤقتة أيضًا عند التعامل مع البيانات الثنائية عند كتابة البرامج منخفضـة المسـتوى مثل التي تتعامل مع إرسال واستقبال البيانات عبر الشـبكة، كمـا تـوفر القـدرة على التعامـل مـع البيانـات على أخفض مستوى ممكن والتعديل عليها في الحالات التي نحتاج بها لذلك.

سنتعرف في هذا الفصل على المخازن المؤقتة وطريقة إنشائها والقراءة والنسخ منها والكتابـة إليهـا، وحـتى تحويل البيانات الثنائية ضمنها إلى صيغ ترميز أخرى.

8.1 إنشاء المخزن المؤقت

سنتعرف في هذه الفقرة على طريقتين لإنشاء كائن التخزين المـؤقت في نـود، حيث يجب يجب أن نسـأل أنفسنا دومًا في ما إذا كنا نريد إنشاء مخزن مؤقت جديد، أو استخراج مخزن مؤقت من بيانات موجودة مسـبقًا، وعلى أساس ذلك سنحدد الطريقة المستخدمة لإنشائه، ففي حال أردنا تخـزين بيانـات غـير موجـودة ونتوقـع أن

تصل لاحقًا ففي تلك الحالة يجب إنشاء مخزن مؤقت جديد باستدعاء بالتابع `alloc()` من الصنف `Buffer`، ولنوضح هذه الطريقة نبدأ بفتح جلسة جديدة من وضع حلقة REPL بتنفيذ الأمر `node` في سطر الأوامر كالتالي:

```
$ node
```

يظهر الرمز `>` في بداية السطر، ما يدل على استعداد هذا الوضع لتلقي التعليمات البرمجية وتنفيذها، حيث يقبل التابع `alloc()` تمرير عدد كمعامل أول إجباري يشير إلى حجم المخزن المؤقت الذي نود إنشاءه، أي يمثل هذا المعامل عدد البايتات التي ستُحجز في الذاكرة للمخزن المؤقت الجديد، فمثلًا لإنشاء مخزن مؤقت بسعة 1 كيلوبايت أي ما يعادل 1024 بايت يمكننا استخدام التابع السابق كالتالي:

```
> const firstBuf = Buffer.alloc(1024);
```

نلاحظ أن الصنف `Buffer` متاح بشكل عام في بيئة نود، ومنه يمكننا الوصول مباشرة إلى التابع `alloc()` لاستخدامه، ونلاحظ كيف مررنا القيمة `1024` كمعامل أول له لينتج لدينا مخزن مؤقت بسعة 1 كيلوبايت، حيث ستحوي المساحة المحجوزة للمخزن المؤقت الجديد مؤقتًا على أصفار افتراضيًا، وذلك ريثما نكتب البيانات ضمنه لاحقًا، وبإمكاننا تخصيص ذلك فإذا أردنا أن تحتوي تلك المساحة على واحدات بدلًا من الأصفار يمكننا تمرير هذه القيمة كمعامل ثاني للتابع `alloc()` كالتالي:

```
> const filledBuf = Buffer.alloc(1024, 1);
```

ينتج لدينا مخزنًا مؤقتًا بمساحة 1 كيلوبايت من الذاكرة المملوءة بالواحدات، ويجب التأكيد أن البيانات التي يمثلها المخزن المؤقت ستكون بيانات ثنائية binary مهما كانت القيمة التي نحددها له كقيمة أولية، حيث يمكن تمثيل العديد من صيغ البيانات بواسطة البيانات الثنائية، فمثلًا البيانات الثنائية التالية تمثل حجم 1 بايت: `01110110`، ويمكن تفسيرها كنص بترميز ASCII باللغة الإنكليزية وبالتالي ستُعبّر عن الحرف `v`، ويمكن أيضًا تفسير هذه البيانات بسياق آخر وترميز مختلف على أنها لون لبكسل واحد من صورة ما، حيث يمكن للحاسوب التعامل مع هذه البيانات ومعالجتها بعد معرفة صيغة ترميزها.

ويستخدم المخزن المؤقت في نود افتراضيًا ترميز UTF-8 في حال كانت القيمة الأولية المخزنة ضمنه عند إنشاءه هي سلسلة نصية، حيث يمكن للبايت الواحد في ترميز UTF-8 أن يمثل حرفًا من أي لغة أو عددًا أو رمزًا ما، ويعتبر هذا الترميز توسعة لمعيار الترميز الأمريكي لتبادل البيانات أو ASCII والذي يقتصر على ترميز الأحرف الإنكليزية الكبيرة والصغيرة والأعداد وبعض الرموز القليلة الأخرى فقط، كعلامة التعجب "!" وعلامة الضم "&"، ويمكننا تحديد الترميز المستخدم من قبل المخزن المؤقت عبر تمريره كمعامل ثالث للتابع `alloc()`، فمثلًا لو اقتصرت حاجة برنامج ما على التعامل مع محارف بترميز ASCII يمكننا تحديده كترميز للبيانات ضمن المخزن المؤقت كالتالي:

```
> const asciiBuf = Buffer.alloc(5, 'a', 'ascii');
```

نلاحظ تمرير المحرف a كمعامل ثانٍ وبذلك سيتم تخزينه ضمن المساحة الأولية التي ستُحجز للمخزن المؤقت الجديد، ويدعم نود افتراضيًا صيغ ترميز المحارف التالية:

- ترميز ASCII ويُمثّل بالسلسلة النصية `ascii`.
- ترميز UTF-8 ويُمثّل بالسلسلة النصية `utf-8` أو `utf8`.
- ترميز UTF-16 ويُمثّل بالسلسلة النصية `utf-16le` أو `utf16le`.
- ترميز UCS-2 ويُمثّل بالسلسلة النصية `ucs-2` أو `ucs2`.
- ترميز Base64 ويُمثّل بالسلسلة النصية `base64`.
- الترميز الست عشري Hexadecimal ويُمثّل بالسلسلة النصية `hex`.
- الترميز ISO/IEC 8859-1 ويُمثّل بالسلسلة النصية `latin1` أو `binary`.

حيث يمكن استخدام أي من أنواع الترميز السابقة مع أي تابع من الصنف `Buffer` يقبل ضمن معاملاته معاملًا بالاسم `encoding` لتحديد صيغة الترميز، ومن ضمنها التابع `alloc()` الذي تعرفنا عليه.

قد نحتاج أحيانًا لإنشاء مخزن مؤقت يُعبر عن بيانات جاهزة موجودة مسبقًا، كقيمة متغير أو سلسلة نصية أو مصفوفة، حيث يمكننا ذلك باستخدام التابع `from()` الذي يدعم إنشاء مخزن مؤقت جديد من عدة أنواع من البيانات وهي:

- مصفوفة من الأعداد التي تتراوح قيمها بين `0` و `255`،حيث يمثل كل عدد منها قيمة بايت واحد.
- كائن من نوع `ArrayBuffer` والذي يخزن داخله حجمًا ثابتًا من البايتات.
- سلسلة نصية.
- مخزن مؤقت آخر.
- أي كائن جافاسكربت يملك الخاصية `Symbol.toPrimitive` التي تُعبر عن طريقة تحويل هذا الكائن إلى بيانات أولية، مثل القيم المنطقية `boolean` أو `null` أو `undefined` أو الأعداد `number` أو السلاسل النصية `string` أو الرموز `symbol`.

لنختبر الآن طريقة إنشاء مخزن مؤقت جديد من سلسلة نصية باستخدام التابع `from` كالتالي:

```
> const stringBuf = Buffer.from('My name is Hassan');
```

ينتج بذلك لدينا كائن مخزن مؤقت جديد يحتوي على قيمة السلسلة النصية `My name is Hassan`، ويمكننا كما ذكرنا إنشاء مخزن مؤقت جديد من مخزن مؤقت آخر مثلًا كالتالي:

```
> const asciiCopy = Buffer.from(asciiBuf);
```

ينتج بذلك لدينا المخزن المؤقت `asciiCopy` والـذي هـو نسـخة مطابقـة من المخـزن الأول `asciiBuf`، وبذلك نكون قد تعرفنا على طرق إنشاء المخازن المؤقتة، وفي الفقرة التالية سنتعلم طرق قراءة البيانات منها.

8.2 القراءة من المخزن المؤقت

يوجد عدة طرق تمكننا من قراءة بيانات المخزن المؤقت، حيث يمكننا قراءة بايت واحد محدد فقط منـه إذا أردنا، أو قراءة كل البيانات دفعة واحدة، ولقراءة بايت واحد فقط يمكن الوصول إليه عـبر رقم تـرتيب مكـان هـذا البايت ضمن المخزن المؤقت، حيث تُخزِّن المخازن المؤقتة البيانات بترتيب متتابع تمامًا كالمصفوفات، ويبـدأ ترتيب أول مكان للبيانات داخلها من الصفر `0` تمامًا كالمصفوفات، ويمكن اسـتخدام نفس صـيغة الوصـول إلى عناصر المصفوفة لقراءة البايتات بشكل مفرد من المخزن مؤقت.

لنختبر ذلك نبدأ بإنشاء مخزن مؤقت جديد من سلسلة نصية كالتالي:

```
> const hiBuf = Buffer.from('Hi!');
```

ونحاول قراءة أول بايت من هذا المخزن كالتالي:

```
> hiBuf[0];
```

بعد الضغط على زر الإدخال ENTER وتنفيذ التعليمة السابقة سيظهر لنا النتيجة التالية:

```
72
```

حيث يرمز العدد `72` ضمن ترميز UTF-8 للحرف `H` وهو أول حرف من السلسلة النصية المُخزنـة، حيث تقـع قيمة أي بايت ضمن المجال من صفر `0` إلى `255`، وذلك لأن البايت يتألف من 8 بتات أو bits، وكـل بت بـدوره يمثل إما صفر `0` أو واحد `1`، فأقصى قيمة يمكن تمثيلها بسلسلة من ثمانية بتات تساوي 2^8 وهو الحجم الأقصى للبايت الواحد، أي يمكن للبايت تمثيل قيمة من 256 قيمة ممكنـة، وبمـا أن أول قيمـة هي الصـفر فـأكبر عـدد يمكن تمثيله في البايت الواحد هو 255، والآن لنحاول قراءة قيمة البايت الثاني ضمن المخزن كالتالي:

```
> hiBuf[1];
```

سـنلاحظ ظهـور القيمـة `105` والـتي ترمـز للحـرف الصـغير `i`، والآن نحـاول قـراءة آخـر بـايت من هـذا المخزن كالتالي:

```
> hiBuf[2];
```

نلاحظ ظهور القيمة 33 والتي ترمز إلى إشارة التعجب ! ولكن ماذا سيحدث لو حاولنا قراءة بايت غير موجود بتمرير قيمة لمكان خاطئ ضمن المخزن كالتالي:

```
> hiBuf[3];
```

سنلاحظ ظهور القيمة التالية:

```
undefined
```

وهو نفس ما سيحدث لو حاولنا الوصول إلى عنصر غير موجود ضمن مصفوفة ما.

والآن بعد أن تعرفنا على طريقة قراءة بايت واحد من البيانات ضمن المخزن مؤقت، سنتعرف على طريقة لقراءة كل البيانات المخزنة ضمنه دفعة واحدة.

يوفر كائن المخزن مؤقت التابعين `toString()` و `toJSON()` والذي يعيد كل منهما البيانات الموجودة ضمن المخزن دفعة واحدة كل منهما بصيغة مختلفة، ونبدأ بالتابع `toString()` والذي يحول البايتات ضمن المخزن المؤقت إلى قيمة سلسلة نصية ويعيدها، لنختبر ذلك باستدعائه على المخزن المؤقت السابق `hiBuf` كالتالي:

```
> hiBuf.toString();
```

سنلاحظ ظهور القيمة التالية:

```
'Hi!'
```

وهي قيمة السلسلة النصية التي خزناها ضمن المخزن المؤقت عند إنشاءه، ولكن ماذا سيحدث لو استدعينا التابع `toString()` على مخزن مؤقت تم إنشاءه من بيانات من نوع مختلف؟ لنختبر ذلك بإنشاء مخزن مؤقت جديد فارغ بحجم 10 بايت كالتالي:

```
> const tenZeroes = Buffer.alloc(10);
```

ونستدعي التابع `toString()` ونلاحظ النتيجة:

```
> tenZeroes.toString();
```

سيظهر ما يلي:

```
'\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000'
```

حيث تقابل السلسلة النصية `\u0000` المحرف في ترميز Unicode المقابل للقيمة `NULL`، وهو ما يقابل قيمة الصفر 0، حيث يعيد التابع `toString()` ترميز UTF-8 للبايتات المخزنة في حال كانت البيانات ضمن

المخزن المؤقت ليست من نوع سلسلة نصية، ويقبل التابع `toString()` معامل اختياري بالاسم `encoding` لتحديد ترميز البيانات المطلوب، حيث يمكن باستخدامه تعديل ترميز قيمة السلسلة النصية التي يعيدها التابع، فيمكن مثلًا قراءة نفس البيانات للمخزن `hiBuf` السابق لكن بالترميز الست عشري كالتالي:

```
> hiBuf.toString('hex');
```

سنلاحظ ظهور النتيجة التالية:

```
'486921'
```

حيث تُعبر تلك القيمة عن الترميز الست عشري للبايتات التي تتألف منها السلسلة النصية `Hi!`.

ويُستفاد في نود من تلك الطريقة لتحويل ترميز بيانات ما من شكل لآخر، بإنشاء مخزن مؤقت جديد يحوي قيمة السلسلة النصية المراد تحويلها ثم استدعاء التابع `toString()` مع تمرير الترميز الجديد المرغوب به.

أما وفي المقابل يعيد التابع `toJSON()` البيانات ضمن المخزن المؤقت كأعداد تمثل قيم البايتات المخزنة مهما كان نوعها، والآن لنختبر ذلك على كل من المخزنين السابقين `hiBuf` و `tenZeroes` ونبدأ بإدخال التعلمية التالية:

```
> hiBuf.toJSON();
```

سنلاحظ ظهور القيمة التالية:

```
{ type: 'Buffer', data: [ 72, 105, 33 ] }
```

يحوي الكائن الناتج من استدعاء التابع `toJSON()` على خاصية النوع `type` بالقيمة نفسها دومًا وهي `Buffer`، حيث يُستفاد من هذه القيمة لتمييز نوع كائن JSON هذا عن الكائنات الأخرى، ويحتوي على خاصية البيانات `data` وهي مصفوفة من الأعداد التي تمثل البايتات المخزنة، ونلاحظ أنها تحتوي على القيم `72` و `105` و `33` بالترتيب وهي نفس القيم التي ظهرت لنا سابقًا عند محاولة قراءة البايتات المخزنة بشكل مفرد.

والآن لنختبر استدعاء التابع `toJSON()` على المخزن الفارغ `tenZeroes`:

```
> tenZeroes.toJSON();
```

سنلاحظ ظهور النتيجة التالية:

```
{ type: 'Buffer', data: [
0, 0, 0, 0,
0, 0, 0, 0
  ] }
```

الخاصية type تحوي نفس القيمة السابقة، بينما البيانات في المصفوفة هي عشرة أصفار تمثل البايتات العشرة الفارغة التي يحويها المخزن المؤقت، وبذلك نكون قد تعلمنا طرق قراءة البيانات من المخازن المؤقتة، وفي الفقرة التالية سنتعلم طريقة التعديل على تلك البيانات ضمن المخزن المؤقت.

8.3 التعديل على المخزن المؤقت

يوجد عدة طرق للتعديل على البيانات ضمن المخزن المؤقت، وهي مشابهة لطريقة قراءة البيانات حيث يمكن إما تعديل قيمة بايت واحد مباشرة باستخدام نفس صيغة الوصول لعناصر المصفوفات، أو كتابة محتوى جديد وتبديل المحتوى المخزن مسبقًا.

ولنبدأ بالتعرف على الطريقة الأولى لذلك سنستخدم المخزن السابق hiBuf الذي يحتوي على قيمة السلسلة النصية Hi! داخله، ولنحاول تعديل محتوى كل بايت منه على حدى إلى أن تصبح القيمة الجديدة هي Hey، حيث نبدأ بتعديل الحرف الثاني من المخزن hiBuf إلى الحرف e كالتالي:

```
> hiBuf[1] = 'e';
```

نتأكد من صحة التعديل السابق بقراءة محتوى المخزن الجديد باستدعاء التابع toString() كالتالي:

```
> hiBuf.toString();
```

نلاحظ ظهور القيمة التالية:

```
'H\u0000!'
```

القيمة الغريبة التي ظهرت تدل على أن المخزن مؤقت يقبل فقط القيم العددية عند تخزينها داخله، لذا لا يمكن تمرير الحرف e كسلسلة نصية مباشرةً، بل يجب تمرير القيمة الثنائية المقابلة له كالتالي:

```
> hiBuf[1] = 101;
```

الآن يمكننا معاينة القيمة الجديدة والتأكد:

```
> hiBuf.toString();
```

نحصل على القيمة التالية:

```
'He!'
```

نعدل الحرف الأخير من هذه القيمة وهو العنصر الثالث ونضع القيمة الثنائية المقابلة للحرف y كالتالي:

```
> hiBuf[2] = 121;
```

نتأكد من المحتوى بعد التعديل:

```
> hiBuf.toString();
```

نحصل على القيمة:

```
'Hey'
```

ماذا سيحدث لو حاولنا تعديل قيمة بـايت يقـع خـارج مجـال بيانـات المخـزن المـؤقت؟ سـنلاحظ تجاهـل المخزن لتلك العملية وتبقى القيمة المخزنة ضمنه كما هي، لنختبر ذلك بكتابة الحرف o إلى المحرف الرابع الغـير موجود ضمن المخزن السابق كالتالي:

```
> hiBuf[3] = 111;
```

نعاين قيمة المخزن بعد ذلك التعديل:

```
> hiBuf.toString();
```

ونلاحظ أن القيمة بقيت كما هي دون تعديل:

```
'Hey'
```

الطريقة الأخرى للتعديل على محتوى المخزن تكون بكتابة عدة بايتات معًا باستخدام التابع write() الـذي يقبل سلسلة نصية كمعامل له تعبر عن المحتـوى الجديـد للبيانـات، لنختـبر ذلـك عـبر تعـديل محتـوى المخـزن hiBuf إلى محتواه السابق Hi! كالتالي:

```
> hiBuf.write('Hi!');
```

نلاحظ أن تنفيذ التعليمة السابقة يعيد القيمة 3 وهي عدد البايتات التي تم تعديلها ضمن المخـزن في تلـك العملية، حيث يعبر كل بايت عن محرف واحد لأننا نسـتخدم الترمـيز UTF-8، وفي حـال كـان المخـزن يسـتخدم ترميز آخر مثل UTF-16 ففيه يُمثَّل كل محرف على 2 بايت، عندها سيعيد تنفيـذ تـابع الكتابـة write() بنفس الطريقة القيمة 6 للدلالة على عدد البايتات التي تمثل المحارف الثلاث المكتوبة.

والآن لنتأكد من المحتوى الجديد بعد التعديل نستدعي toString() كالتالي:

```
> hiBuf.toString();
```

نحصل على القيمة:

```
'Hi!'
```

هذه الطريقة أسرع من طريقة تعديل كل بايت على حدى، ولكن ماذا سيحدث لو كتبنا بيانات بحجم أكبر من حجم المخزن الكلي؟ سيقبل المخزن البيانات المقابلة لحجمه فقط ويهمل البقية، لنختبر ذلك بإنشاء مخزن مؤقت بحجم 3 بايت كالتالي:

```
> const petBuf = Buffer.alloc(3);
```

ونحاول كتابة سلسلة نصية بأربعة محارف مثلًا `Cats` كالتالي:

```
> petBuf.write('Cats');
```

نلاحظ أن ناتج التعليمة السابقة هي القيمة `3` أي تم تعديل قيمة ثلاث بايتات فقط وتجاهل باقي القيمة المُمررة، لنتأكد من القيمة الجديدة كالتالي:

```
> petBuf.toString();
```

نلاحظ القيمة الجديدة:

```
'Cat'
```

حيث يُعدل التابع `write()` البايتات بالترتيب فعدّل أول ثلاث بايتات فقط ضمن المخزن وتجاهل البقية.

والآن لنختبر ماذا سيحدث لو كتبنا قيمة بحجم أقل من حجم المخزن الكلي، لهذا نُنشئ مخزن مؤقت جديد بحجم 4 بايت كالتالي:

```
> const petBuf2 = Buffer.alloc(4);
```

ونكتب القيمة الأولية داخله كالتالي:

```
> petBuf2.write('Cats');
```

ثم نكتب قيمة جديدة حجمها أقل من حجم المخزن الكلي كالتالي:

```
> petBuf2.write('Hi');
```

وبما أن البيانات ستكتب بالترتيب بدءًا من أول بايت سنلاحظ نتيجة ذلك عند معاينة القيمة الجديدة للمخزن:

```
> petBuf2.toString();
```

ليظهر القيمة التالية:

```
'Hits'
```

تم تعديل قيمة أول بايتين فقط، وبقيت البايتات الأخرى كما هي دون تعديل.

تكون البيانات التي نود كتابتها موجودة أحيانًا ضمن مخزن مؤقت آخر، حيث يمكننا في تلك الحالة نسخ محتوى ذلك المخزن باستدعاء التابع copy()، لنختبر ذلك بداية بإنشاء مخزنين جديدين كالتالي:

```
> const wordsBuf = Buffer.from('Banana Nananana');
> const catchphraseBuf = Buffer.from('Not sure Turtle!');
```

يحوي كل من المخزنين wordsBuf و catchphraseBuf على بيانات من نوع سلسلة نصية، فإذا أردنا تعديل قيمة المخزن catchphraseBuf ليحوي على القيمة Nananana Turtle! بدلًا من Not sure Turtle! يمكننا استدعاء تابع النسخ copy() لنسخ القيمة Nananana من المخزن wordsBuf إلى catchphraseBuf، حيث نستدعي التابع copy() على المخزن الحاوي على المعلومات المصدر لنسخها إلى مخزن آخر، ففي مثالنا النص الذي نريد نسخه موجود ضمن المخزن wordsBuf، لذا نستدعي تابع النسخ منه كالتالي:

```
> wordsBuf.copy(catchphraseBuf);
```

حيث يُعبّر معامل الوجهة target المُمرر له عن المخزن المؤقت الذي ستُنسخ البيانات إليه، ونلاحظ ظهور القيمة 15 كنتيجة لتنفيذ التعليمة السابقة وهي تعبر عن عدد البايتات التي تم كتابتها، ولكن بما أن النص Nananana مكوّن من ثمانية محارف فقط فهذا يدل على عمل مختلف نفذه تابع النسخ، لنحاول معرفة ماذا حدث ونعاين القيمة الجديدة باستخدام التابع toString() ونلاحظ النتيجة:

```
> catchphraseBuf.toString();
```

نلاحظ القيمة الجديدة:

```
'Banana Nananana!'
```

نلاحظ أن تابع النسخ copy() قد نسخ كامل المحتوى من المخزن wordsBuf وخزنه ضمن catchphraseBuf، ولكن ما نريده هو نسخ قسم من تلك البيانات فقط وهي القيمة Nananana، لنعيد القيمة السابقة للمخزن catchphraseBuf أولًا ثم نحاول تنفيذ المطلوب كالتالي:

```
> catchphraseBuf.write('Not sure Turtle!');
```

يقبل التابع copy() عدة معاملات تمكننا من تحديد البيانات التي نرغب بنسخها إلى المخزن المؤقت الوجهة وهي:

- الوجهة target وهو المعامل الإجباري الوحيد، ويعبر عن المخزن المؤقت الوجهة لنسخ البيانات.

- `targetStart` وهو ترتيب أول بايت ستبدأ كتابة البيانات إليه ضمن المخزن الوجهة، وقيمته الافتراضية هي الصفر `0`، أي بدء عملية الكتابة من أول بايت ضمن المخزن الوجهة.
- `sourceStart` وهو ترتيب أول بايت من البيانات التي نرغب بنسخها من المخزن المصدر.
- `sourceEnd` وهو ترتيب آخر بايت من البيانات الذي ستتوقف عملية النسخ عنده في المخزن المصدر، وقيمته الافتراضية هي الطول الكلي للبيانات ضمن المخزن المصدر.

باستخدام تلك المعاملات يمكننا تحديد الجزء `Nananana` من المخزن `wordsBuf` ليُنسخ إلى المخزن `catchphraseBuf`، حيث نمرر المخزن `catchphraseBuf` كمعامل الوجهة `target` كما فعلنا سابقًا، ونمرر القيمة `0` للمعامل `targetStart` لكتابة القيمة `Nananana` في بداية المخزن `catchphraseBuf`، أما للقيمة `sourceStart` سنمرر `7` وهو ترتيب بداية أول محرف من القيمة `Nananana` ضمن المخزن `wordsBuf`، وللقيمة `sourceEnd` نمرر الحجم الكلي للمخزن المصدر، ليكون الشكل النهائي لاستدعاء تابع النسخ بعد تخصيص المعاملات السابقة كالتالي:

```
> wordsBuf.copy(catchphraseBuf, 0, 7, wordsBuf.length);
```

سيظهر هذه المرة القيمة `8` كنتيجة لتلك العملية ما يعني أن القيمة التي حددناها فقط هي ما تم نسخه، ونلاحظ كيف استخدمنا الخاصية `wordsBuf.length` لتمرير حجم المخزن كقيمة للمعامل `sourceEnd`، وهي نفس الخاصية `length` الموجودة ضمن المصفوفات، والآن لنعاين القيمة الجديدة للمخزن `catchphraseBuf` ونتأكد من النتيجة:

```
> catchphraseBuf.toString();
```

نلاحظ القيمة الجديدة:

```
'Nananana Turtle!'
```

بذلك نكون قد عدلنا البيانات ضمن المخزن `catchphraseBuf` عن طريق نسخ جزء محدد من بيانات المخزن `wordsBuf` إليه.

والآن بعد أن انتهينا من تنفيذ الأمثلة في هذا الفصل يمكنك الخروج من جلسة REPL حيث ستُحذف كل المتغيرات السابقة التي عرفناها بعد عملية الخروج هذه، ولذلك ننفذ أمر الخروج كالتالي:

```
> .exit
```

8.4 خاتمة

تعرفنا في هذا الفصل على المخازن المؤقتة والتي تمثل مساحة محددة من الذاكرة محجوزة لتخزين البيانات بالصيغة الثنائية، وتعلمنا طرق إنشاء المخازن المؤقتة، سواء الجديدة أو التي تحتوي على بيانات موجودة مسبقًا، وتعرفنا بعدها على طرق قراءة تلك البيانات من المخزن سواء بقراءة كل بايت منه على حدى أو قراءة المحتوى كاملًا باستخدام التابعين `toString()` و `toJSON()`، ثم تعرفنا على طرق الكتابة إلى المخازن لتعديل البيانات المخزنة ضمنها، سواء بكتابة كل بايت على حدى أو باستخدام التابعين `write()` و `copy()`.

يفتح التعامل مع المخازن المؤقتة في نود Node.js الباب للتعامل مع البيانات الثنائية مباشرة، فيمكن مثلًا دراسة تأثير صيغ الترميز المختلفة للمحارف على البيانات المخزنة، كمقارنة صيغ الترميز المختلفة مع الصيغتين UTF-8 و ASCII وملاحظة فرق الحجم بينها، كما يمكن مثلًا تحويل البيانات المخزنة من صيغة UTF-8 إلى صيغ الترميز الأخرى، ويمكنك الرجوع إلى التوثيق الرسمي العربي من نود للكائن Buffer للتعرف عليه أكثر.

9. استخدام مرسل الأحداث Event emitter

مرسل أو مطلق الأحداث event emitter هو كائن في نود Node.js مهمته إطلاق حدث ما عبر إرسال رسالة تخبر بوقوع حدث، حيث يمكن استخدامه لربط تنفيذ بعض التعليمات البرمجية في جافاسكربت بحدث ما، وذلك عبر الاستماع لذلك الحدث وتنفيذ تابع ما عند كل تنبيه بحدوثه، ويتم تمييز تلك الأحداث عن بعضها بسلسلة نصية تُعبّر عن اسم الحدث ويمكن إرفاق بيانات تصف ذلك الحدث إلى التوابع المُستمعة له.

عادة ما نربط تنفيذ التعليمات البرمجية بعد اكتمال حدث ما باستخدام طرق البرمجة اللامتزامنة asynchronous programming، كتمرير توابع رد النداء أو ربط الوعود مع بعضها، ولكن من مساوئ تلك الطرق هو الربط بين أمر تنفيذ الحدث والتعليمات الواجب تنفيذها بعد انتهاءه، مما يزيد صعوبة التعديل على تلك التعليمات لاحقًا، وهنا يأتي دور مرسل الأحداث ليوفر طريقة بديلة للربط بين الحدث والمهام المرتبطة به، باتباع نمط ناشر-مشترك publish-subscribe، حيث يرسل فيه الناشر أو مرسل الأحداث رسالة تعبر عن حدث ما، ثم يستقبل بدوره المشترك هذه الإشارة وينفذ تعليمات برمجية استجابة لذلك الحدث، ومن ميزات هذا النمط هو الفصل بين الناشر والمشترك، بحيث لا يعلم الناشر أي شيء عن المشتركين، فينشر الناشر الرسائل فقط ثم يتفاعل معها المشتركون كلٌّ بطريقته الخاصة، وبالتالي يصبح تعديل التطبيق أسهل عبر تعديل طريقة عمل المشتركين فقط دون أي تعديل على الناشر.

سنتعلم في هذا الفصل طريقة إنشاء واستخدام مرسل الأحداث عبر تطوير صنف مرسل أحداث خاص لإدارة شراء البطاقات بالاسم `TicketManager`، وسنربط به بعض المشتركين الذين سيتفاعلون مع حدث الشراء `buy` الذي سيُنشر بعد كل عملية شراء لبطاقة ما، وسنتعلم أيضًا طرقًا لمعالجة أحداث الأخطاء التي قد يرسلها المرسل، وكيفية إدارة المشتركين بالأحداث.

9.1 إرسال أحداث Emitting Events

سنتعلم في هذه الفقرة طريقتين لإنشاء مرسل أحداث في نود، الأولى باستخدام صنف مرسل الأحداث مباشرةً `EventEmitter`، والثانية بإنشاء صنف خاص يرث من صنف مرسل الأحداث الأساسي، ويعتمد الاختيار بين هاتين الطريقتين على مدى الترابط بين الأحداث ضمن التطبيق وبين العمليات التي ستسبب إرسالها، فإذا كانت العمليات داخل الكائن هي ما ستسبب إرسال الأحداث، أي يوجد ترابط وثيق بين العمليات والأحداث فهنا يفضل استخدام طريقة الوراثة من صنف مرسل الأحداث الأساسي، أما إذا كان العمليات منفصلة أو متفرقة، مثلًا نتيجة عدة عمليات نُفذت ضمن أكثر من كائن، فيفضل استخدام كائن مرسل للأحداث منفصل نستخدمه ضمن التطبيق داخليًا.

ولنبدأ بالتعرف على طريقة استخدام كائن مرسل أحداث منفصل، ونبدأ أولًا بإنشاء مجلد للمشروع بالاسم `event-emitters` كالتالي:

```
$ mkdir event-emitters
```

وندخل إلى المجلد:

```
$ cd event-emitters
```

نُنشئ ملف جافاسكربت جديد بالاسم firstEventEmitter.js ونفتحه ضمن أي محرر نصوص، حيث سنستخدم في أمثلتنا محرر `nano` كالتالي:

```
$ nano firstEventEmitter.js
```

يمكن استخدام الصنف `EventEmitter` الموجود ضمن الوحدة `events` في نود لإرسال الأحداث، ولنبدأ باستيراد ذلك الصنف من تلك الوحدة كالتالي:

```
const { EventEmitter } = require("events");
```

ثم ننشئ كائنًا جديدًا من ذلك الصنف:

```
const { EventEmitter } = require("events");

const firstEmitter = new EventEmitter();
```

ونختبر إرسال حدث ما من هذا الكائن كالتالي:

```
const { EventEmitter } = require("events");

```

```
const firstEmitter = new EventEmitter();


firstEmitter.emit("My first event");
```

نلاحظ استدعاء التابع `emit()` لإرسال حدث جديد، حيث نمرر له اسم ذلك الحدث كسلسلة نصية وبعـدها يمكن تمرير أي عدد من المعاملات الخاصة بذلك الحدث، حيث تفيد تلك المعاملات بإرسال بيانات إضافية مع الحدث تتلقاها التوابع المستمعة للحدث وتوفر بيانات إضافية توصف ذلك الحدث، وسنستخدم ذلك في مثالنا لاحقًا عندما نرسل حدث شراء لبطاقة جديدة بتمرير بعض البيانات المتعلقة بعملية الشراء تلك، ويجب أن نمـيز اسم الحدث لأننا سنستخدمه لاحقًا كما هو للاستماع إليه.

> يعيد تنفيذ تابع الإرسال `emit()` قيمة منطقية تكون صحيحة `true` في حال كان هناك أي تابع يستمع لذلك الحدث، وفي حال لم يكن هناك أي مستمع سيعيد القيمة `false` رغم عدم توفر معلومات أخرى عن المستمعين.

والآن نحفظ الملف وننفذه باستخدام الأمر `node` ونلاحظ النتيجة:

```
$ node firstEventEmitter.js
```

نلاحظ عدم ظهور أي خرج من عملية التنفيذ السابقة، وذلك لأننا لم نطبع أي رسالة إلى الطرفية ولا يوجـد أي مشتركين يستمعون للحدث المرسل.

والآن لنبدأ بتطبيق مثال مدير شراء البطاقات، حيث سيوفر هذا الصنف تابعًا لعملية الشراء وبعـد أن إتمـام هذه العملية بنجاح سيُرسل حدث يعبر عن ذلك مرفقًا ببيانات حول المشتري للبطاقة، ثم سنطور وحـدة برمجيـة منفصلة لمحاكاة عملية إرسال بريد إلكتروني للمشتري استجابة لحدث الشراء لنعلمه بنجاح العملية.

نبدأ بإنشاء مدير البطاقات حيث سيرث صنف مرسل الأحداث الأساسـي `EventEmitter` مباشـرة كي لا نضـطر لإنشـاء كـائن مرسـل للأحـداث منفصـل داخليًـا واسـتخدامه، ونُنشـئ ملـف جافاسـكربت جديد بالاسم ticketManager.js:

```
$ nano ticketManager.js
```

كما فعلنا سابقًا نستورد الصنف `EventEmitter` من الوحدة `events` لاستخدامه كالتالي:

```
const EventEmitter = require("events");
```

ونعرف صنف مدير البطاقات `TicketManager` الذي سيوفر تابع الشراء لاحقًا:

```
const EventEmitter = require("events");
```

```
class TicketManager extends EventEmitter {}
```

نلاحظ أن صنف مدير البطاقات `TicketManager` يرث من صنف مرسل الأحداث الأساسي `EventEmitter` ما يعني أنه سيرث كل التوابع والخواص التي يوفرها صنف مرسل الأحداث وبالتالي يمكننا استدعاء تابع إرسال الأحداث `emit()` من الصنف نفسه مباشرةً.

ولنبدأ بتعريف التابع الباني للصنف لتمرير كمية البطاقات المتوفرة للبيع، والذي سيُستدعى عند إنشاء كائن جديد من هذا الصنف كالتالي:

```
const EventEmitter = require("events");

class TicketManager extends EventEmitter {
    constructor(supply) {
        super();
        this.supply = supply;
    }
}
```

يقبل التابع الباني معامل العدد `supply` والذي يعبر عن الكمية المتوفرة للبيع، وبما أن الصنف `TicketManager` يرث من صنف مرسل الأحداث الأساسي `EventEmitter` فيجب استدعاء التابع الباني للصنف الأب عبر استدعاء `super()` وذلك لتهيئة توابع وخاصيات الصنف الأب بشكل صحيح.

وبعد ذلك نعرف قيمة خاصية الكمية `supply` ضمن الصنف بواسطة `this.supply` ونسند القيمة المُمررة للتابع الباني لها، والآن سنضيف تابع شراء بطاقة جديدة `buy()` حيث سيُنقص هذا التابع كمية البطاقات المتوفرة ويرسل حدثًا يحوي تفاصيل عملية الشراء كالتالي:

```
const EventEmitter = require("events");

class TicketManager extends EventEmitter {
    constructor(supply) {
        super();
        this.supply = supply;
    }

    buy(email, price) {
        this.supply--;
```

```
            this.emit("buy", email, price, Date.now());
        }
    }
```

نلاحظ تمرير عنوان البريد الإلكتروني والعنوان الخاص بالمشتري والسعر المدفوع ثمنًا للبطاقة للتابع buy()، حيث سينقص التابع كمية البطاقات المتوفرة بمقدار واحد، ثم سيرسل حدث الشراء buy مع تمرير بيانات إضافية هذه المرة وهي عنوان البريد الإلكتروني للمشتري وسعر البطاقة وتوقيت عملية الشراء تلك.

والآن ولكي تستطيع باقي الوحدات البرمجية استخدام هذا الصنف يجب تصديره في نهاية الملف كالتالي:

```
...

module.exports = TicketManager
```

نحفظ الملف ونخرج منه، ونكون بذلك انتهينا من إعداد صنف مدير البطاقات المُرسل للأحداث TicketManager، وأصبح جاهزًا لإرسال الأحداث المتعلقة بعملية شراء البطاقات الجديدة وبقي علينا الاشتراك والاستماع لذلك الحدث ومعالجته، وهذا ما سنتعرف عليه في الفقرة التالية حيث سنُنشئ توابع تستمع لذلك الحدث.

9.2 الاستماع للأحداث

يمكن تسجيل مستمع إلى حدث ما باستدعاء التابع on() من كائن مرسل الأحداث، حيث سيستمع لحدث معين وعند إرساله سيستدعي لنا تابع رد النداء المُمرر له، وصيغة استدعاءه كالتالي:

```
eventEmitter.on(event_name, callback_function) {
    action
}
```

> التابع on() هو اسم بديل للتابع addListener() ضمن مرسل الأحداث ولا فرق في استخدام أي منهما، حيث سنستخدم في أمثلتنا التابع on() دومًا.

والآن لنبدأ بالاستماع إلى الأحداث بإنشاء ملف جافاسكربت جديد بالاسم firstListener.js:

```
$ nano firstListener.js
```

سنختبر عملية تسجيل المستمع بطباعة رسالة ضمنه إلى الطرفية عند تلقي الحدث، ونبدأ باستيراد الصنف TicketManager ضمن الملف الجديد كالتالي:

```
const TicketManager = require("./ticketManager");
```

```
const ticketManager = new TicketManager(10);
```

مررنا القيمة 10 للصنف TicketManager كقيمة لمخزون البطاقات المتاحة، والآن لنضيف مستمع جديد لحدث الشراء buy كالتالي:

```
const TicketManager = require("./ticketManager");

const ticketManager = new TicketManager(10);

ticketManager.on("buy", () => {
    console.log("Someone bought a ticket!");
});
```

لإضافة مستمع جديد نستدعي التابع on() من الكائن ticketManager، والمتوفر ضمن كل كائنات صنف مرسل الأحداث، وبما أن الصنف TicketManager يرث من صنف مرسل الأحداث الأساسي EventEmitter بالتالي فهذا التابع أصبح متوفرًا ضمن أي كائن من صنف مدير البطاقات TicketManager.

نمرر تابع رد نداء للتابع on() كمعامل ثاني حيث ستنفذ التعليمات ضمنه عند كل إطلاق للحدث، حيث يطبع هذا التابع الرسالة "Someone bought a ticket!" إلى الطرفية عند كل حدث لعملية الشراء buy.

وبعد أن سجلنا التابع كمستمع للحدث ننفذ عملية الشراء باستدعاء التابع buy() لينتج عنه إرسال لحدث الشراء كالتالي:

```
...

ticketManager.buy("test@email.com", 20);
```

استدعينا تابع الشراء buy بعنوان البريد الإلكتروني test@email.com وبسعر 20 لتلك للبطاقة، والآن نحفظ الملف ونخرج منه وننفذ البرنامج بتنفيذ الأمر node كالتالي:

```
$ node firstListener.js
```

نلاحظ ظهور الخرج:

```
Someone bought a ticket!
```

بذلك يكون مرسل الأحداث قد أرسل الحدث بنجاح وتم معالجته من قبل تابع الاستماع.

والآن لنجرب أكثر من عملية شراء ونراقب ماذا سيحدث، نفتح الملف firstListener.js للتعديل مجددًا ونستدعي تابع الشراء buy() مرة أخرى:

```
...

ticketManager.buy("test@email.com", 20);
ticketManager.buy("test@email.com", 20);
```

نحفظ الملف ونخرج منه وننفذ الملف مجددًا ونلاحظ النتيجة هذه المرة:

```
Someone bought a ticket!
Someone bought a ticket!
```

بما أن تابع الشراء buy() قد استُدعي مرتين فقد نتج عنه إرسال لحدث الشراء buy مرتين أيضًا، ثم استقبل تابع الاستماع هذين الحدثين وطبع الرسالة مرتين.

قد نحتاج في بعض الأحيان للاستماع لأول مرة يُرسَل فيها الحدث فقط وليس لكل مرة، ويمكن ذلك عبر استدعاء تابع مشابه للتابع on() وهو once() يعمل بنفس الطريقة، فهو سيسجل تابع الاستماع للحدث المحدد بالمعامل الأول، وسينفذ التابع المُمرر كمعامل ثاني له، ولكن الفرق هنا أن التابع once() وبعد استقبال الحدث لأول مرة سيُلغي اشتراك تابع الاستماع بالحدث ويزيله، وينفذه لمرة واحدة فقط عند أول استقبال للحدث بعد عملية التسجيل.

ولنوضح ذلك باستخدامه ضمن الملف firstListener.js نفتحه مجددًا للتعديل ونضيف في نهايته الشيفرة التالية لتسجيل تابع للاستماع لحدث الشراء لمرة واحدة فقط باستخدام once() كالتالي:

```
const TicketManager = require("./ticketManager");
const ticketManager = new TicketManager(10);

ticketManager.on("buy", () => {
        console.log("Someone bought a ticket!");
});

ticketManager.buy("test@email.com", 20);
ticketManager.buy("test@email.com", 20);

ticketManager.once("buy", () => {
    console.log("This is only called once");
```

```
});
```

نحفظ الملف ونخرج منه وننفذ البرنامج ونلاحظ الخرج التالي:

```
Someone bought a ticket!
Someone bought a ticket!
```

لا نلاحظ أي فرق هذه المرة عن الخرج السابق، والسبب أننا سجلنا مستمع للحدث بعد الانتهاء من إرسال حدث الشراء buy وليس قبله، لذا لم يُنفذ التابع لأنه لم يستقبل أي أحداث جديدة، أي لا يمكن الاستماع إلا للأحداث التي سترد لاحقًا بعد عملية تسجيل التابع، أما الأحداث السابقة فلا يمكن الاستماع لها، ولحل المشكلة يمكن استدعاء تابع الشراء buy() مرتين من جديد بعد تسجيل تابع الاستماع باستخدام once() لنتأكد أنه لن يُنفذ سوى لمعالجة أول حدث منها فقط:

```
...

ticketManager.once("buy", () => {
    console.log("This is only called once");
});

ticketManager.buy("test@email.com", 20);
ticketManager.buy("test@email.com", 20);
```

نحفظ الملف ونخرج منه وننفذ البرنامج لنحصل على خرج كالتالي:

```
Someone bought a ticket!
Someone bought a ticket!
Someone bought a ticket!
This is only called once
Someone bought a ticket!
```

أول رسالتين ظهرتا نتيجة أول استدعاءين لتابع الشراء buy() وقبل تسجيل تابع الاستماع باستخدام once()، ولكن إضافة تابع الاستماع الجديد لا يزيل وجود التوابع المسجلة سابقًا، وستبقى تستمع للأحداث اللاحقة وتطبع تلك الرسائل، وبوجود توابع استماع تم تسجيلها باستخدام on() قبل تابع الاستماع الجديد الذي سجلناه باستخدام once()، فسنلاحظ ظهور الرسالة Someone bought a ticket! قبل الرسالة This is only called once، وكلا السطرين هما استجابة لحدث الشراء buy الثاني وما بعده.

وعند آخر استدعاء لتابع الشراء `buy()` لم يبقى ضمن مرسل الأحداث سوى التوابع التي تستمع لهذا الحدث والتي سُجلت باستخدام `on()`، حيث أن التابع المستمع الذي سجلناه باستخدام `once()` أزيل تلقائيًا بعد تنفيذه لمرة واحدة فقط.

وبذلك نكون قد تعلمنا الطرق المختلفة لتسجيل توابع الاستماع للأحداث، وسنتعلم في الفقرة التالية كيف يمكننا الوصول للبيانات المرسلة مع الأحداث لمعالجتها.

9.3 استقبال بيانات الحدث

تعلمنا في الفقرة السابقة طريقة الاستماع للأحداث والاستجابة لها، لكن عادة ما يُرسل مع هذه الأحداث بيانات إضافية توضح الحدث، وسنتعلم في هذه الفقرة كيف يمكننا استقبال البيانات والتعامل معها.

سننشئ وحدتين برمجيتين الأولى لإرسال البريد الإلكتروني والثانية لتسجيل البيانات في قاعدة البيانات، ولن نتطرق لتفاصيل تلك الوحدات بل سنضع مثالًا يُعبّر عن تنفيذ العمليات الخاصة بها لأن تركيزنا هو على طريقة استقبالها لبيانات الأحداث، وبعد إنشاء تلك الوحدتين سنربطهما مع مرسل الأحداث ضمن ملف البرنامج الأساسي index.js.

والآن لنبدأ بإنشاء وحدة خدمة إرسال البريد الإلكتروني البرمجية نُنشئ لها ملف جافاسكربت جديد ونفتحه ضمن محرر النصوص:

```
$ nano emailService.js
```

ستحوي هذه الوحدة على صنف يوفر تابع الإرسال `send()` والذي سنمرر له عنوان البريد الإلكتروني المأخوذ من بيانات حدث الشراء buy كالتالي:

```
class EmailService {
    send(email) {
        console.log(`Sending email to ${email}`);
    }
}

module.exports = EmailService
```

عرفنا الصنف `EmailService` الحاوي على تابع الإرسال `send()` والذي بدلًا من إرسال بريد إلكتروني حقيقي سيطبع رسالة توضح تنفيذ هذه العملية مع توضيح عنوان البريد الإلكتروني المرسل إليه.

والآن نحفظ الملف ونخرج منه ثم نُنشئ ملف جافاسكربت جديد بالاسم databaseService.js لوحدة خدمة قاعدة البيانات البرمجية ونفتحه ضمن محرر النصوص:

```
$ nano databaseService.js
```

سيُحاكي هذا الصنف حفظ بيانات عملية الشراء ضمن قاعدة البيانات عند استدعاء تابع الحفظ save() كالتالي:

```
class DatabaseService {
    save(email, price, timestamp) {
        console.log(`Running query: INSERT INTO orders VALUES (email,
price, created) VALUES (${email}, ${price}, ${timestamp})`);
    }
}

module.exports = DatabaseService
```

عرفنا الصنف DatabaseService الحاوي على تابع الحفظ save() حيث سيحاكي عملية حفظ البيانات إلى قاعدة البيانات بطباعة البيانات الممررة له إلى الطرفية أيضًا، حيث سنمرر له البيانات المرفقة مع حدث الشراء buy وهي عنوان البريد الإلكتروني للمشتري وسعر البطاقة وتوقيت عملية الشراء.

والآن نحفظ الملف ونخرج منه ونبدأ بربط مدير البطاقات TicketManager مع كل من خدمتي البريد الإلكتروني EmailService وخدمة قاعدة البيانات DatabaseService، حيث سنسجل تابع استماع لحدث الشراء buy سيستدعي داخله تابع إرسال البريد الإلكتروني send() وتابع حفظ البيانات في قاعدة البيانات save()، لذا ننشئ ملف جافاسكربت الرئيسي للبرنامج index.js ونفتحه ضمن محرر النصوص ونبدأ باستيراد الوحدات البرمجية اللازمة:

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");
```

ثم ننشئ كائنات جديدة من الأصناف السابقة، وفي هذه الخطوة سنحدد كمية قليلة للبطاقات المتاحة كالتالي:

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");

const ticketManager = new TicketManager(3);
const emailService = new EmailService();
```

```
const databaseService = new DatabaseService();
```

بعدها نبدأ بتسجيل تابع الاستماع لحدث الشراء باستخدام الكائنات السابقة، حيث نريد بعد كل عملية شراء لبطاقة جديدة إرسال بريد إلكتروني للمشتري وحفظ بيانات تلك العملية في قاعدة البيانات، لذلك نضيف ما يلي:

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");

const ticketManager = new TicketManager(3);
const emailService = new EmailService();
const databaseService = new DatabaseService();


ticketManager.on("buy", (email, price, timestamp) => {
    emailService.send(email);
    databaseService.save(email, price, timestamp);
});
```

أضفنا كما تعلمنا سابقًا تابع استماع للحدث باستخدام التابع `on()`، والفرق هذه المرة أننا نقبل ثلاث معاملات ضمن تابع رد النداء تمثل البيانات المرفقة مع الحدث، ولمعرفة البيانات التي سترسل نعاين طريقة إرسال حدث الشراء داخل التابع `buy()` من صنف مدير البطاقات:

```
this.emit("buy", email, price, Date.now());
```

حيث سيقابل كل معامل نقبله ضمن تابع رد النداء معاملًا من البيانات التي نمررها لتابع إرسال الحدث السابق، فأول معامل هو البريد الإلكتروني `email` ثم السعر `price` ثم توقيت الشراء وهو التوقيت الحالي `Date.now()` والذي يقابل المعامل الأخير المسمى `timestamp` في تابع رد النداء.

وفي تابع الاستماع للحدث وعند كل إرسال لحدث الشراء `buy` سيُستدعى تابع إرسال البريد الإلكتروني `send()` من كائن الخدمة `emailService`، ثم تابع حفظ البيانات ضمن قاعدة البيانات من كائن الخدمة الخاصة به `databaseService`.

والآن لنختبر عملية الربط تلك كاملة باستدعاء تابع الشراء `buy()` في نهاية الملف:

```
...
```

```
ticketManager.buy("test@email.com", 10);
```

نحفظ الملف ونخرج منه، ننفذ البرنامج بتنفيذ الأمر node ونعاين النتيجة:

```
$ node index.js
```

نلاحظ ظهور النتيجة التالية:

```
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588720081832)
```

نلاحظ استقبال تابع الاستماع للبيانات المرفقة بالحدث بنجاح، والآن بعد أن تعلمنا طرق إضافة توابع الاستماع لمختلف الأحداث بأسماء وبيانات مختلفة ماذا عن الأخطاء التي قد تحدث خلال عملية الشراء؟ وكيف يمكننا معالجتها والاستجابة لها؟ هذا ما سنتعرف عليه في الفقرة التالية، حيث سنتعرف أيضًا على المعايير الواجب اتباعها عند معالجة الأخطاء.

9.4 معالجة أخطاء الأحداث

عند فشل تنفيذ عملية ما يجب أن يُعلم مرسل الأحداث المشتركين بذلك، والطريقة المتبعة عادةً في نود تكون بإرسال حدث مخصص بالاسم error يُعبّر عن حدوث خطأ ما أثناء التنفيذ، ويرفق به كائن الخطأ Error لتوضيح المشكلة.

وحاليًا في صنف مدير البطاقات لدينا الكمية المتاحة تتناقص بمقدار واحد في كل مرة ننفذ تابع الشراء buy()، ويمكن حاليًا تجاوز الكمية المتاحة وشراء عدد غير محدود من البطاقات، لنحل هذه المشكلة بتعديل تابع الشراء buy() ليرسل حدث يعبر عن خطأ في حال نفاذ الكمية المتاحة من البطاقات ومحاولة أحدهم شراء بطاقة جديدة، لذا نعود لملف مدير البطاقات ticketManager.js ونعدل تابع الشراء buy() ليصبح كالتالي:

```
...

buy(email, price) {
    if (this.supply > 0) {
        this.supply--;
        this.emit("buy", email, price, Date.now());
        return;
    }
```

```
    this.emit("error", new Error("There are no more tickets left to
purchase"));
}
...
```

أضفنا العبارة الشرطية `if` لحصر عملية شراء البطاقات فقط في حال توفر كمية منها، عبر التحقق من أن الكمية الحالية أكبر من الصفر، أما في حال نفاذ الكمية سنرسل حدث الخطأ `error` ونرفق به كائن خطأ `Error` جديد يحوي وصفًا حول سبب الخطأ.

والآن نحفظ الملف ونخرج منه ونحاول الوصول لتلك الحالة من الملف الرئيسي index.js، فحاليًا نشتري بطاقة واحدة فقط والكمية المتاحة ضمن كائن مدير البطاقات `ticketManager` هي ثلاث بطاقات فقط، لذا للوصول لحالة الخطأ يجب أن نشتري أربعة بطاقات لتجاوز الكمية المتاحة، لهذا نعود للملف index.js لنعدل عليه ونضيف الأسطر التالية في نهاية الملف لشراء أربعة بطاقات:

```
...

ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
```

نحفظ الملف ونخرج منه وننفذ البرنامج:

```
$ node index.js
```

نحصل على خرج التالي:

```
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588724932796)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588724932812)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588724932812)
events.js:196
      throw er; // Unhandled 'error' event
      ^
```

```
Error: There are no more tickets left to purchase
    at TicketManager.buy
(/home/hassan/event-emitters/ticketManager.js:16:28)
    at Object.<anonymous> (/home/hassan/event-emitters/index.js:17:15)
    at Module._compile (internal/modules/cjs/loader.js:1128:30)
    at Object.Module._extensions..js
(internal/modules/cjs/loader.js:1167:10)
    at Module.load (internal/modules/cjs/loader.js:983:32)
    at Function.Module._load (internal/modules/cjs/loader.js:891:14)
    at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:71:12)
    at internal/main/run_main_module.js:17:47
Emitted 'error' event on TicketManager instance at:
    at TicketManager.buy
(/home/hassan/event-emitters/ticketManager.js:16:14)
    at Object.<anonymous> (/home/hassan/event-emitters/index.js:17:15)
    [... lines matching original stack trace ...]
    at internal/main/run_main_module.js:17:47
```

جرى معالجة أول ثلاث أحداث شراء buy بنجاح بينما سبب الحدث الرابع بعد نفاذ الكمية ذلك الخطأ، لنعاين رسالة الخطأ:

```
...
events.js:196
      throw er; // Unhandled 'error' event
      ^

Error: There are no more tickets left to purchase
    at TicketManager.buy
(/home/hassan/event-emitters/ticketManager.js:16:28)
...
```

توضح رسالة الخطأ نتيجة التنفيذ ونلاحظ تحديدًا رسالة الخطأ التالية:

```
"Unhandled 'error' event"
```

والتي تعني أن خطأ الحدث لم يتم معالجته، ما يعني أنه في حال أرسل مرسل الأحداث حدث الخطأ ولم نسجل أي مستمع لمعالجة هذا الحدث سيتم رمي الخطأ كما بالشكل السابق، ما يؤدي كما رأينا لتوقف تنفيذ

البرنامج، لذا يفضل دومًا الاستماع لحدث الخطأ `error` من مرسل الأحداث لحل هذه المشكلة ومعالجة هذا الحدث لمنع توقف عمل البرنامج.

والآن لنطبق ذلك ونضيف تابع لمعالجة حدث الخطأ ضمن الملف index.js حيث نضيف تابعًا مستمعًا لحدث الخطأ قبل تنفيذ عملية شراء البطاقات، وذلك لأنه وكما ذكرنا سابقًا لا يمكن سوى معالجة الأحداث التي سترد منذ لحظة تسجيل تابع الاستماع، لذا نضيف تابع معالجة الخطأ كالتالي:

```
...

ticketManager.on("error", (error) => {
    console.error(`Gracefully handling our error: ${error}`);
});

ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
```

نطبع داخل ذلك التابع رسالة إلى الطرفية تدل على معالجة الخطأ المُرسل باستخدام `console.error()`، والآن نحفظ الملف ونخرج منه ثم نعيد تنفيذ البرنامج لنرى ما إذا كانت معالجة الخطأ ستتم بنجاح:

```
$ node index.js
```

لنحصل على الخرج التالي هذه المرة:

```
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588726293332)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588726293348)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email.com, 10, 1588726293348)
Gracefully handling our error: Error: There are no more tickets left
to purchase
```

نلاحظ في آخر سطر ظهور رسالة معالجة الخطأ من قبل تابع الاستماع الذي سجلناه ولم يفشل تنفيذ البرنامج كما حدث سابقًا.

والآن وبعد أن تعلمنا طرق إرسال والاستماع للأحداث بمختلف أنواعها سنتعرف في الفقرة التالية على طرق مفيدة لإدارة توابع الاستماع للأحداث.

9.5 إدارة توابع الاستماع للأحداث

يوفر صنف مرسل الأحداث طرقًا لمراقبة عدد توابع الاستماع المشتركة بحدث ما والتحكم بها، حيث يمكن مثلًا الاستفادة من التابع `listenerCount()` لمعرفة عدد توابع الاستماع المسجلة لحدث معين ضمن الكائن، حيث يقبل ذلك التابع معامل يدل على الحدث الذي نريد معرفة عدد المستمعين له.

والآن لنعود للملف الأساسي index.js ونطبق ذلك حيث نزيل بدايةً استدعاءات تابع الشراء `buy()` الأربعة السابقة ثم نضيف السطرين التاليين لتصبح الشيفرة كالتالي:

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");

const ticketManager = new TicketManager(3);
const emailService = new EmailService();
const databaseService = new DatabaseService();

ticketManager.on("buy", (email, price, timestamp) => {
    emailService.send(email);
    databaseService.save(email, price, timestamp);
});

ticketManager.on("error", (error) => {
    console.error(`Gracefully handling our error: ${error}`);
});

console.log(`We have ${ticketManager.listenerCount("buy")} listener(s)
for the buy event`);
console.log(`We have ${ticketManager.listenerCount("error")}
listener(s) for the error event`);
```

بدلًا من استدعاء تابع الشراء `buy()` نطبع إلى الطرفية سطران الأول لطباعة عدد التوابع المُستمعة لحدث الشراء buy باستخدام التابع `listenerCount()`، والثاني لطباعة عدد التوابع المستمعة لتابع الخطأ `error`، والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج مجددًا باستخدام الأمر node لنحصل على الخرج التالي:

```
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
```

بما أننا استدعينا تابع التسجيل `on()` مرة واحدة لحدث الشراء `buy` ومرة واحدة أيضًا لحدث الخطأ `error` فالخرج السابق صحيح.

سنستفيد من التابع `listenerCount()` عندما نتعلم طريقة إزالة توابع الاستماع من مرسل الأحداث لنتأكد من عدم وجود مشتركين بحدث ما، فقد نحتاج أحيانًا لتسجيل تابع استماع لفترة مؤقتة فقط ثم نزيله بعد ذلك.

يمكن الاستفادة من التابع `off()` لإزالة تابع استماع من كائن مرسل الأحداث، ويقبل معاملين هما اسم الحدث وتابع الاستماع الذي نرغب بإزالته.

> التابع `off()` هو اسم بديل عن التابع `removeListener()` وكلاهما ينفذ نفس العملية ويقبل نفس المعاملات، وسنستخدم في أمثلتنا التابع `off()` دومًا.

وبما أنه يجب تمرير تابع الاستماع الذي نرغب بإزالته كمعامل ثانٍ للتابع `off()` فيجب حفظ ذلك التابع أولًا ضمن متغير أو ثابت كي نشير إليه لاحقًا ونمرره للإزالة، فلا تصلح طريقة استخدام التوابع التي سجلناها سابقًا للأحداث `buy` و `error` للإزالة باستخدام `off()`.

ولنتعرف على طريقة عمل تابع الإزالة `off()` سنضيف تابع استماع جديد ونختبر إزالته، ونبدأ بتعريف تابع رد النداء وحفظه ضمن متغير سنمرره لاحقًا لتابع الإزالة `off()`، والآن نعود للملف الأساسي index.js ونفتحه ضمن محرر النصوص ونضيف التالي:

```
...

const onBuy = () => {
    console.log("I will be removed soon");
};
```

بعدها نسجل هذا التابع للاستماع إلى الحدث `buy` كالتالي:

```
...

ticketManager.on("buy", onBuy);
```

وللتأكد من تسجيل التابع بشكل سليم سنطبع عدد التوابع المستمعة للحدث `buy` ثم نستدعي تابع الشراء `buy()`:

```
...


console.log(`We added a new event listener bringing our total count
for the buy event to: ${ticketManager.listenerCount("buy")}`);
ticketManager.buy("test@email", 20);
```

نحفظ الملف ونخرج منه ونشغل البرنامج:

```
$ node index.js
```

سيظر لنا الخرج التالي:

```
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
We added a new event listener bringing our total count for the buy
event to: 2
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email, 20, 1588814306693)
I will be removed soon
```

نلاحظ ظهور الرسالة التي توضح عدد التوابع المستمعة لذلك الحدث، ثم استدعينا بعدها التابع `buy()` ونلاحظ تنفيذ تابعي الاستماع لذلك الحدث، حيث نفذ المستمع الأول عمليتي إرسال البريد الإلكتروني وحفظ البيانات ضمن قاعدة البيانات، ثم طبع المستمع الثاني الرسالة `I will be removed soon`.

والآن لنختبر إزالة تابع الاستماع الثاني باستخدام `off()`، لذا نعود للملف مجددًا ونضيف عملية الإزالة بواسطة `off()` في نهاية الملف وبعدها نطبع عدد توابع الاستماع الحالية المسجلة لنتأكد من الإزالة، ثم نختبر استدعاء تابع الشراء `buy()` مجددًا:

```
...


ticketManager.off("buy", onBuy);


console.log(`We now have: ${ticketManager.listenerCount("buy")}
listener(s) for the buy event`);
ticketManager.buy("test@email", 20);
```

نلاحظ كيف مررنا المتغير `onBuy` كمعامل ثاني لتابع الإزالة `off()` لنحدد تابع الاستماع الذي نرغب بإزالته، نحفظ الملف ونخرج منه وننفذ البرنامج لمعاينة النتيجة:

```
$ node index.js
```

نلاحظ أن الخرج بقي كما كان سابقًا، وظهر سطر جديد يؤكد عملية الإزالة ويوضح عدد التوابـع المسـجلة، ثم بعد استدعاء تابع الشراء buy() نلاحظ ظهور خرج تابع الاستماع الأول فقط، بينما أُزيل تابع الاستماع الثاني:

```
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
We added a new event listener bringing our total count for the buy
event to: 2
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email, 20, 1588816352178)
I will be removed soon
We now have: 1 listener(s) for the buy event
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, created)
VALUES (test@email, 20, 1588816352178)
```

يمكن أيضًا إزالة كل توابع الاستماع لحدث ما دفعة واحـدة باسـتدعاء التـابع removeAllListeners()، ونمرر له اسم الحدث الذي نرغب بإزالة التوابع التي تستمع إليه، وسنستفيد من هذا التابع لإزالـة تـابع الاسـتماع الأول للحدث buy الذي لم نتمكن من إزالته سابقًا بسبب طريقة تعريفه

الآن نعود للملف index.js ونزيل كافة توابـع الاسـتماع باسـتخدام removeAllListeners() ثم نطبـع عدد التوابع المسجلة باستخدام listenerCount() للتأكد من نجاح العملية، ونتحقق من ذلـك أيضًـا بتنفيـذ عملية شراء جديدة بعد الإزالة، ونلاحظ أن لا شيء سيحدث بعد إرسال ذلك الحدث، لذا نضيف الشـيفرة التاليـة في نهاية الملف:

```
...

ticketManager.removeAllListeners("buy");
console.log(`We have ${ticketManager.listenerCount("buy")} listeners
for the buy event`);
ticketManager.buy("test@email", 20);
console.log("The last ticket was bought");
```

نحفظ الملف ونخرج منه ونشغل البرنامج:

```
$ node index.js
```

نحصل على الخرج:

```
...

ticketManager.removeAllListeners("buy");
console.log(`We have ${ticketManager.listenerCount("buy")} listeners
for the buy event`);
ticketManager.buy("test@email", 20);
console.log("The last ticket was bought");
```

نلاحظ بعد إزالة كل توابع الاستماع لم يُرسل أي بريد إلكتروني ولم تُحفظ أي بيانات في قاعدة البيانات.

9.6 خاتمة

تعلمنا في هذا الفصل وظيفة مرسل الأحداث وطريقة إرسال الأحداث منه باستخدام التابع `emit()` الذي يوفره الصنف `EventEmitter`، ثم تعلمنا طرق الاستماع لتلك الأحداث باستخدام التابعين `on()` و `once()` لتنفيذ التعليمات البرمجية استجابة لإرسال حدث ما، وتعلمنا كيف يمكن معالجة أحداث الأخطاء، وكيفية مراقبة توابع الاستماع المسجلة باستخدام `listenerCount()`، وإدارتها باستخدام التابعين `off()` و `removeAllListeners()`.

لو استخدمنا توابع رد النداء callbacks والوعود promises للاستجابة للأحداث ضمن نظام مدير البطاقات لكنا سنحتاج لربطه مع الوحدات البرمجية للخدمات الأخرى كخدمة البريد الإلكتروني وخدمة قاعدة البيانات، لكن بالاستفادة من مرسل الأحداث تمكنا من فصل تلك الوحدات البرمجية عن بعضها، ويمكن لأي وحدة برمجية جديدة قد نضيفها لاحقًا وتستطيع الوصول لمدير البطاقات أن تُربط معه وتستجيب للأحداث التي سيرسلها، وهي الفائدة التي يوفرها التعامل مع مرسل الأحداث فعند تطوير وحدة برمجية نرغب بربطها لاحقًا مع عدة وحدات برمجية أخرى أو مراقبتها يمكن أن نجعلها ترث صنف مرسل الأحداث الأساسي لتسهيل عملية الربط تلك.

ويمكنك الرجوع إلى توثيق نود الرسمي العربي لمرسل الأحداث للتعرف عليه أكثر.

10. تنقيح الأخطاء باستخدام المنقح debugger وأدوات المطور DevTools

عملية تتبع أخطاء البرامج لمعرفة مصدر المشكلة في نود Node.js خلال مرحلة التطـوير تـوفر على المطـور الكثير من وقت تطوير المشروع، وتزداد صعوبة تلك المهمة مع كبر حجم المشروع وزيادة تعقيده، وهنا يأتي دور مُنقِّح الأخطاء debugger ليساعد في ذلك، وهو برنامج يسمح للمطور بمعاينة البرنامج أثناء تشغيله عـبر تنفيـذ الشيفرة سطرًا تلو الآخر ومعاينة حالة التطبيق وتغيرها، مما يوفر للمبرمج نظرة أقرب على طريقـة عمـل البرنـامج ما يسهل العثور على الأخطاء وإصلاحها.

وعادة ما يضيف المطورون تعليمات الطباعة داخل شيفرة البرنامج لمعاينة بعض القيم أثناء تشـغيله، حيث يضيف المطور في نود تعليمات طباعة مثل `console.log()` و `console.debug()`، ومع أن هذه الطريقة سهلة وسريعة لكنها تبقى يدوية ولا تخدم دومًا في الحالات المعقدة أو عندما يكون التطـبيق كبـيرًا، فقـد ينسـى أحيانًا المطور بعض تعليمات الطباعة تلك ما قد يؤدي لطباعة معلومـات خاصـة وحساسـة عن التطـبيق يجعلـه عرضة للاختراق، وهنا يوفر لنا المنقح طريقة أفضل لمراقبة البرنامج أثناء التشغيل دون أن يُعرّض البرنامج لمثـل تلك الأخطار.

وأهم ميزتين في منقح الأخطاء الداخلي هما مراقبة الكائنات، وإضافة نقـاط الوقـوف breakpoints، حيث تتيح مراقبة الكائنات طريقة لمشاهدة التغير في حالة المتغيرات أثناء تنفيـذ البرنـامج خطـوة بخطـوة، أمـا نقـاط الوقوف فهي أماكن ضمن الشيفرة يمكن للمبرمج تحديدها ليتوقـف البرنـامج عن التنفيـذ مؤقتًـا عنـد الوصـول إليها، ليعطي فرصة للمبرمج لمعاينة حالة البرنامج في تلك اللحظة.

سنتعلم في هذا الفصل طريقة استخدام المنقح لاستكشـاف الأخطـاء ضـمن بعض الـبرامج في نـود، حيث سنستخدم بدايةً أداة تنقيح الأخطاء الداخليـة في نـود ونتعلم طريقـة إعـداد المراقبـة للمتغـيرات وإضـافة نقـاط التوقف لنتمكن من اكتشاف المشاكل وإصلاحها، ثم سنتعلم استخدام واجهـة أداة المطـور في متصـفح جوجـل كروم بدلًا من التعامل مع المنقح من سطر الأوامر.

انتبه إلى أنك ستحتاج إلى تثبيت متصفح جوجل كروم أو متصفح كروميوم مفتوح المصدر في هذا الفصل.

10.1 استخدام الراصدات Watchers مع المنقح Debugger

الميزتين الأساسيتين لمنقح الأخطاء هما مراقبة المتغيرات وتغير قيمها أثناء التنفيذ، وميزة الإيقاف المؤقت لعمل البرنامج عند أماكن محددة من الشيفرة باستخدام نقاط الوقوف، وسنتعلم في هذه الفقرة طريقة مراقبة المتغيرات لتساعدنا في اكتشاف الأخطاء.

تساعدنا عملية مراقبة المتغيرات ورصدها في فهم كيفية تغير قيم تلك المتغيرات أثناء تنفيذ البرنامج، وسنستفيد من هذه الميزة في اكتشاف الأخطاء في منطق عمل البرنامج وإصلاحها، وسنبدأ بإنشاء مجلد جديد بالاسم debugging سيحوي على البرامج التي سنتعامل معها:

```
$ mkdir debugging
```

وندخل إلى المجلد:

```
$ cd debugging
```

ننشئ داخله ملف جافاسكربت جديد بالاسم badLoop.js ونفتحه ضمن أي محرر نصوص، حيث سنستخدم في أمثلتنا محرر نانو nano كالتالي:

```
$ nano badLoop.js
```

سنكتب برنامجًا يمر على عناصر المصفوفة ويجمع قيمها لحساب المجموع الكلي لها، حيث تمثل تلك الأرقام عدد الطلبات اليومي لمتجر خلال فترة أسبوع، حيث سيطبع البرنامج المجموع الكلي للأرقام في تلك المصفوفة، ليكون البرنامج كالتالي:

```
let orders = [341, 454, 198, 264, 307];

let totalOrders = 0;

for (let i = 0; i <= orders.length; i++) {
  totalOrders += orders[i];
}

console.log(totalOrders);
```

أنشأنا بداية مصفوفة الطلبات `orders` والتي تحوي خمسة أعداد، ثم أنشأنا متغير المجموع الكلي للطلبات `totalOrders` وضبطنا قيمته الأولية إلى الصفر `0`، حيث سنخزن ضمنه المجموع الكلي للأرقام

السابقة، ومررنا ضمن حلقة for على عناصر المصفوفة orders وأضفنا كـل قيمـة منهـا إلى متغـير المجمـوع الكلي totalOrders، ثم أخيرًا طبعنا قيمة المجموع الكلي.

والآن نحفظ الملف ونخرج منه وننفذ البرنامج ونعاين النتيجة:

```
$ node badLoop.js
```

يظهر لنا الخرج التالي:

```
NaN
```

القيمة NaN في جافاسكربت هي اختصار لجملة "ليس عددًا" أو "Not a Number"، ولكن كيـف حصـلنا على تلك القيمة مع أن المصفوفة لا تحـوي سـوى قيم عدديـة؟ الطريقـة الأفضـل لمعرفـة سـبب المشـكلة هي استخدام منقح الأخطاء، وهنا سنبدأ بالتعرف على منقح نود ونستخدمه رصد قيمة كل من المتغـيرين totalOr ders و i ضمن حلقة for، ولتشغيله نضيف خيـار inspect قبـل اسـم الملـف عنـد تشـغيله بواسـطة الأمر node كالتالي:

```
$ node inspect badLoop.js
```

سنلاحظ ظهور الخرج التالي:

```
< Debugger listening on ws://127.0.0.1:9229/e1ebba25-04b8-410b-811e-
8a0c0902717a
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in badLoop.js:1
> 1 let orders = [341, 454, 198, 264, 307];

let totalOrders = 0;
```

يحتوي السطر الأول من الخـرج على رابـط خـادم تنقيح الأخطـاء، حيث تسـتفيد منـه أدوات تنقيح الأخطـاء الخارجية مثل متصفح الويب للتواصل مع خادم التنقيح الخاص بنود وهو مـا سـنتعامل معـه لاحقًـا، وافتراضـيًا يكـون هـذا الخـادم متاحًـا على المنفـذ 9229: والعنـوان المحلي localhost أو 127.0.0.1، ويفضـل منـع الوصول لهذا المنفذ من الشبكة الخارجية والوصول إليه من الجهاز محليًا فقط.

وبعد ربط منقح الأخطاء ستظهر الرسالة Break on start in badLoop.js:1 والتي تعني توقف التنفيذ عند أول سطر من الملف، حيث يمكن وضع نقاط الوقوف ضمن الشيفرة لتحديـد مكـان توقـف التنفيـذ وكما لاحظنا فمنقح الأخطاء يتوقف افتراضيًا عند أول سطر من الملف دومًا ويُظهر لنا مقطـع من الشـيفرة عنـد مكان التوقف وبعده سطر جديد يبدأ بالكلمة debug يمكننا كتابة الأوامر ضمنه:

```
...
> 1 let orders = [341, 454, 198, 264, 307];

let totalOrders = 0;
debug>
```

نلاحظ وجود الرمز > بجانب رقم السطر الأول 1 وهو دلالة على مكان توقف التنفيذ الحالي، ويُظهر السطر الأخير استعداد منقح الأخطاء لتلقي الأوامر، حيث يمكننا مثلًا تنفيذ أمر لتوجيهه لتقديم عملية التنفيذ خطوة إلى الأمام والذهاب إلى السطر التالي من التنفيذ، ويمكن إدخال أحد الأوامر التالية:

- c أو cont: لإكمال عملية التنفيذ حتى الوصول إلى نقطة الوقوف التالية أو حتى الانتهاء من تنفيذ البرنامج.
- n أو next: للتقدم خطوة إلى الأمام في التنفيذ إلى السطر التالي من الشيفرة.
- s أو step: للدخول إلى دالة ما، حيث تكون عملية التقدم افتراضيًا ضمن النطاق scope الذي نصحح الأخطاء ضمنه فقط، وتمكننا هذه العملية من الدخول ضمن دالة استدعتها الشيفرة التي نفحصها لمعاينة عملها من الداخل ومراقبة تعاملها مع البيانات المُمررة لها.
- o: للخروج من دالة حيث سيعود التنفيذ لخارجها إلى مكان استدعائها، وهو المكان الذي ستُرجع قيمة تنفيذ الدالة إليه، حيث يفيد هذا الأمر في العودة مباشرةً إلى خارج الدالة إلى المكان الذي كنا نعاينه قبل الدخول إليها.
- pause: لإيقاف التنفيذ مباشرةً مؤقتًا.

لنتقدم بتنفيذ البرنامج سطرًا تلو الآخر بتنفيذ الأمر n للانتقال إلى السطر التالي:

```
debug> n
```

نلاحظ تقدم التنفيذ إلى السطر الثالث:

```
break in badLoop.js:3
let orders = [341, 454, 198, 264, 307];

> 3 let totalOrders = 0;

for (let i = 0; i <= orders.length; i++) {
```

يتم تجاوز الأسطر الفارغة، لذا إذا قدّمنا علمية التنفيذ سطرًا آخر الآن بتنفيذ الأمر n مجددًا سينتقل التنفيذ إلى السطر الخامس:

```
break in badLoop.js:5
let totalOrders = 0;

> 5 for (let i = 0; i <= orders.length; i++) {
totalOrders += orders[i];
}
```

يوقف التنفيذ الآن في بداية الحلقة، وإذا كانت الطرفية تدعم إظهار الألوان في الخرج سنلاحظ تحديد القيمة `0` ضمن التعليمة `let i = 0`، حيث يحدد المنقح أي قسم من الشيفرة على وشك التنفيذ، ففي الحلقة `for` أول ما ينفذ هو إسناد القيمة لعداد الحلقة، وسنبدأ هنا بمعاينة القيم للمتغيرات لنحدد سبب الحصول على القيمة `NaN` بدلًا من القيمة العددية لمتغير المجموع `totalOrders`، حيث أن قيمتي المتغيرين `totalOrders` و `i` تتغيران عند كل دورة للحلقة، وسنستفيد من ميزة الرصد والمراقبة التي يوفرها المنقح في مراقبة قيم هذين المتغيرين.

نبدأ بإعداد المراقبة لمتغير المجموع الكلي `totalOrders` بتنفيذ التعليمة التالية:

```
debug> watch('totalOrders')
```

لمراقبة أي متغير خلال تنقيح الأخطاء نستدعي الدالة `watch()` الذي يوفرها المنقح ونمرر لها سلسلة نصية تحوي على اسم المتغير الذي نريد مراقبته، وبعد الضغط على زر الإدخال ENTER وتنفيذ الدالة `watch()` سينتقل التنفيذ إلى سطر جديد دون ظهور أي خرج، وستظهر القيم التي نراقبها عند الانتقال للسطر التالي.

لنراقب أيضًا المتغير الآخر `i` بنفس الطريقة:

```
debug> watch('i')
```

سنشاهد الآن عملية المراقبة للمتغيرات السابقة، ننفذ الأمر `n` للانتقال خطوة للأمام وسيظهر لنا التالي:

```
break in badLoop.js:5
Watchers:
totalOrders = 0
i = 0

let totalOrders = 0;

> 5 for (let i = 0; i <= orders.length; i++) {
totalOrders += orders[i];
}
```

نلاحظ ظهور قيم المتغيرين اللذين نراقبهما `totalOrders` و `i` قبل الشيفرة حيث سيتم تحديث هذه القيم عند تغيرها، ونلاحظ أن المنقح يحدد حاليًا الخاصية `length` من التعليمة `orders.length`، ما يعني أن الخطوة التالية هي التحقق من شرط إكمال التنفيذ للحلقة قبل إعادة تنفيذ التعليمات في جسم الحلقة، وبعدها ستُنفذ تعليمة زيادة قيمة عداد الحلقة `i++`.

والآن نتقدم خطوة للأمام بتنفيذ الأمر `n` مجددًا للدخول إلى جسم الحلقة:

```
break in badLoop.js:6
Watchers:
totalOrders = 0
i = 0


for (let i = 0; i <= orders.length; i++) {
> 6   totalOrders += orders[i];
}
  8
```

ستُعدِّل التعليمة الحالية من قيمة المتغير `totalOrders`، وسنلاحظ ذلك من تغير تلك القيمة ضمن قسم المراقبة في الأعلى.

والآن نتقدم خطوة إلى الأمام بتنفيذ `n` ليظهر لنا ما يلي:

```
Watchers:
totalOrders = 341
i = 0

let totalOrders = 0;

> 5 for (let i = 0; i <= orders.length; i++) {
totalOrders += orders[i];
}
```

نلاحظ أن قيمة متغير المجموع الكلي `totalOrders` تساوي قيمة أول عنصر من المصفوفة 341، والخطوة التالية الآن هي التحقق من شرط إكمال تنفيذ الحلقة، لذا ننفذ الأمر `n` لتعديل قيمة عداد الحلقة `i`:

```
break in badLoop.js:5
Watchers:
```

```
totalOrders = 341
i = 1

let totalOrders = 0;

> 5 for (let i = 0; i <= orders.length; i++) {
totalOrders += orders[i];
}
```

إلى الآن قد تقدمنا عدة خطوات يدويًا ضمن الشيفرة لمراقبة التغير في قيم المتغيرات، لكن تلك الطريقة غير عملية حيث سنتعرف في الفقرة التالية على حل لهذه المشكلة باستخدام نقاط الوقوف، وسنُكمل حاليًا العمل بتقديم عملية التنفيذ يدويًا ومراقبة قيم المتغيرات للعثور على سبب المشكلة.

والآن نتقدم في التنفيذ 12 خطوة للأمام لنلاحظ الخرج التالي:

```
break in badLoop.js:5
Watchers:
totalOrders = 1564
i = 5

let totalOrders = 0;

> 5 for (let i = 0; i <= orders.length; i++) {
totalOrders += orders[i];
}
```

عدد القيم ضمن المصفوفة `orders` هو خمسة، ولكن قيمة عداد الحلقة `i` الحالية هي `5`، وبما أننا نستخدم قيمة المتغير `i` للوصول إلى العنصر ضمن المصفوفة بالترتيب الحالي فالقيمة عند الترتيب `orders[5]` غير موجودة، وترتيب آخر قيمة ضمن المصفوفة `orders` هو `4`، ما يعني أن محاولة الوصول للعنصر السادس باستخدام `orders[5]` سيعيد القيمة `undefined`.

والآن نتقدم بالتنفيذ خطوة للأمام بتنفيذ الأمر `n`:

```
break in badLoop.js:6
Watchers:
totalOrders = 1564
i = 5
```

```
for (let i = 0; i <= orders.length; i++) {
> 6   totalOrders += orders[i];
}
  8
```

وبالتقدم خطوة إضافية بتنفيذ `n` نلاحظ القيمة الجديدة للمتغير `totalOrders`:

```
break in badLoop.js:5
Watchers:
totalOrders = NaN
i = 5

let totalOrders = 0;

> 5 for (let i = 0; i <= orders.length; i++) {
totalOrders += orders[i];
}
```

لاحظنا بالاستفادة من عملية تنقيح الشيفرة ومراقبة قيم المتغيرين `totalOrders` و `i` أن الحلقة تُنفَّذ ستة مرات بدلًا من خمسة، وعندما تكون قيمة عداد الحلقة `i` هي `5` فمحاولة الوصول للعنصر الحالي `orders[5]` وإضافته للمتغير `totalOrders` ستجعل من قيمة المجموع تساوي `NaN`، لأن قيمة العنصر السادس `orders[5]` الغير موجود ستكون `undefined`، فإذًا المشكلة هي في شرط الحلقة `for` فبدلًا من التحقق من أن قيمة العداد `i` هي أصغر أو تساوي طول المصفوفة `orders` يجب أن نتحقق من أنها أصغر من الطول فقط.

وبعد أن حددنا المشكلة نخرج من المنقح ونصحح الخطأ ضمن الشيفرة ونعيد تنفيذ البرنامج ونتحقق من النتيجة، لكن أولًا ننفذ أمر الخروج `.exit` ثم نضغط زر الإدخال ENTER:

```
debug> .exit
```

نخرج بذلك من وضع المنقح ونعود إلى الملف badLoop.js ونفتحه ضمن محرر النصوص ونعدل شرط حلقة `for` كالتالي:

```
...
for (let i = 0; i < orders.length; i++) {
...
```

نحفظ الملف ونخرج منه ونشغل البرنامج:

```
$ node badLoop.js
```

سنلاحظ ظهور قيمة المجموع الصحيحة ونكون بذلك حللنا المشكلة:

```
1564
```

نكون بذلك قد تعلمنا طريقة استخدام المنقح ودالة مراقبة المتغيرات watch الخاصة به لاستكشاف وتحديد الأخطاء أثناء التنفيذ!

وسنتعلم الآن في الفقرة التالية كيف يمكننا الاستفادة من نقاط الوقوف لتنقيح الأخطاء ضمن البرنامج دون الحاجة لتقديم التنفيذ يدويًا سطرًا تلو الآخر.

10.2 استخدام نقاط الوقوف Breakpoints

تتألف البرامج في نود عادة من عدة وحدات برمجية يتشابك عملها مع بعضها بعضًا، لذا محاولة تنقيح الأخطاء سطرًا تلو الآخر كما فعلنا في الفقرة السابقة أمر صعب وغير مجدي في التطبيقات الكبيرة المعقدة، وهنا يأتي دور نقاط الوقوف breakpoints لحل تلك المشكلة.

تسمح نقاط الوقوف بتخطي التنفيذ إلى السطر الذي نريده مباشرةً وإيقاف البرنامج لمعاينة حالته آنذاك، حيث لإضافة نقطة وقوف في نود نضيف الكلمة المحجوزة debugger ضمن الشيفرة مباشرةً، ويمكننا بعدها وخلال عملية التنقيح التنقل بين نقاط الوقوف ضمن الشيفرة بتنفيذ الأمر c في طرفية التنقيح بدلًا من الأمر n السابق، ويمكننا إضافة المراقبة للتعليمات التي نرغب بها عند نقاط الوقوف تلك.

سنتعرف على طريقة استخدام نقاط الوقوف بمثال عن برنامج يقرأ قائمة من الجمل ويستخرج منها الكلمة الأكثر تكرارًا ويعيدها لنا، لذلك سنُنشئ لهذا المثال ثلاث ملفات، الأول هو ملف يحوي الجمل النصية sentences.txt التي سيعالجها البرنامج، حيث سنضيف داخله كمثال أول فقرة من مقال عن سمكة قرش الحوت من موسوعة بريتانيكا Britannica بعد إزالة علامات الترقيم منها، لذلك ننشئ الملف ونفتحه ضمن محرر النصوص:

```
$ nano sentences.txt
```

ونكتب داخله النص التالي:

```
Whale shark Rhincodon typus gigantic but harmless shark family
Rhincodontidae that is the largest living fish

Whale sharks are found in marine environments worldwide but mainly in
tropical oceans
```

```
They make up the only species of the genus Rhincodon and are
classified within the order Orectolobiformes a group containing the
carpet sharks

The whale shark is enormous and reportedly capable of reaching a
maximum length of about 18 metres 59 feet

Most specimens that have been studied however weighed about 15 tons
about 14 metric tons and averaged about 12 metres 39 feet in length

The body coloration is distinctive

Light vertical and horizontal stripes form a checkerboard pattern on a
dark background and light spots mark the fins and dark areas of the
body
```

نحفظ الملف ونخرج منه، ونضيف الشيفرة التالية إلى ملف جافاسكربت جديد بالاسم textHelper.js، حيث سيحوي هذا الملف على بعض الدوال المساعدة في معالجة الملف النصي السابق خلال عملية تحديد الكلمة الأكثر تكرارًا من النص، ونبدأ بإنشاء الملف textHelper.js ونفتحه ضمن محرر النصوص:

```
$ nano textHelper.js
```

ونضيف ثلاث دوال لمعالجة النص ضمن الملف sentences.txt الأول لقراءة الملف:

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

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  return sentences;
};
```

نستورد الوحدة البرمجية `fs` من نود لنتمكن من قراءة الملف، بعدها نضيف الدالة `readFile()` التي تستخدم التابع `readFileSync()` لتحميل محتوى الملف sentences.txt ككائن مخزن مؤقت `Buffer` ثم تستدعي منه التابع `toString()` لتحويل المحتوى إلى سلسلة نصية.

نضيف بعدها دالة لتجزئة السلسلة نصية السابقة إلى مصفوفة من الكلمات كالتالي:

```
...

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
```

```
  words = words.map((word) => word.trim().toLowerCase());
  return words;
};
```

استفدنا من التوابع `split()` و `join()` و `map()` لتحويل السلسلة النصية إلى مصفوفة من الكلمات الموجودة ضمنها، وحولنا حالة كل كلمة منها إلى أحرف صغيرة لتسهيل عملية المقارنة بينها وإحصائها.

أما الدالة الثالثة والأخيرة فستحصي تكرار كل كلمة ضمن مصفوفة الكلمات السابقة ويعيد كل الكلمات مع تكراراتها ضمن كائن يعبر عن النتيجة كالتالي:

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  return map;
};
```

أنشأنا كائنًا جديدًا بالاسم `map` يحوي الكلمات ضمن النص كمفاتيح وعدد مرات تكرارها كقيم لها، ثم مررنا على عناصر مصفوفة الكلمات وأضفناها إلى ذلك الكائن إن تكن موجودة أو زدنا قيمة تكرارها قيمة واحدة.

وأخيرًا لنصدر تلك الدوال لنتمكن من استخدامها ضمن الوحدات البرمجية الأخرى:

```
...

module.exports = { readFile, getWords, countWords };
```

نحفظ الملف ونخرج منه، والآن سننشئ الملف الثالث والأخير ضمن المثال هو الملف الأساسي الذي سيستعين بالدوال ضمن الوحدة البرمجية السابقة textHelper.js لاستخراج أكثر كلمة تكرارًا من النص.

نبدأ بإنشاء الملف index.js ثم نفتحه ضمن محرر النصوص:

```
$ nano index.js
```

نستورد الوحدة البرمجية textHelpers.js كالتالي:

```
const textHelper = require('./textHelper');
```

وننشئ مصفوفة جديدة تحتوي على بعض الكلمات المكررة الشائعة التي نرغب بتجاهلها مثل حروف العطف والجر والضمائر وبعض الصفات، تدعى الكلمات الشائعة أو stop words:

```
...


const stopwords = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours',
'ourselves', 'you', 'your', 'yours', 'yourself', 'yourselves', 'he',
'him', 'his', 'himself', 'she', 'her', 'hers', 'herself', 'it', 'its',
'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what',
'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is',
'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had',
'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and',
'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at',
'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through',
'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up',
'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further',
'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all',
'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such',
'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too',
'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now', ''];
```

بهذه الطريقة سنحصل على كلمات ذات معاني من ضمن النص الذي نعالجه بدلًا من الحصول على كلمات مثل أدوات التعريف التي تتكرر كثيرًا مثل the و a.

نبدأ باستخدام الدوال المساعدة من الوحدة textHelper.js لقراءة النص واستخراج الكلمات منه وإحصاء مرات التكرار لكل منها كالتالي:

```
...


let sentences = textHelper.readFile();
let words = textHelper.getWords(sentences);
let wordCounts = textHelper.countWords(words);
```

بعد ذلك سنستخرج أكثر كلمة تكرارًا منها، وخوارزمية تحديد الكلمة الأكثر تكرارًا هي بالمرور أولًا على مفاتيح كائن الكلمات المحصاة ومقارنة التكرار مع آخر أعلى قيمة مررنا عليها سابقًا، وفي حال كانت قيمة التكرار للمفتاح الحالي أعلى من الكلمة السابقة سنحدد تكرار الكلمة الحالية على أنه التكرار الأعلى، لتصبح الشيفرة لهذه الخوارزمية كالتالي:

```
...

let max = -Infinity;
let mostPopular = '';

Object.entries(wordCounts).forEach(([word, count]) => {
  if (stopwords.indexOf(word) === -1) {
    if (count > max) {
      max = count;
      mostPopular = word;
    }
  }
});

console.log(`The most popular word in the text is "${mostPopular}"
with ${max} occurrences`);
```

استخدمنا التابع `Object.entries()` لتحويل المفاتيح والقيم ضمن الكائن `wordCounts` إلى مصفوفة، ثم استخدمنا التابع `forEach()` وداخله عبارة شرطية لاختبار قيمة التكرار للكلمة الحالية مع أعلى قيمة تكرار شاهدناها سابقًا.

والآن نحفظ الملف ونخرج منه وننفذه كالتالي:

```
$ node index.js
```

نلاحظ ظهور النتيجة التالية:

```
The most popular word in the text is "whale" with 1 occurrences
```

لكن الجواب الذي ظهر خاطئ فنلاحظ تكرار الكلمة `whale` أكثر من مرة ضمن النص في الملف sentences.txt، وهذه المرة قد يكون السبب في أحد الدوال العديدة المستخدمة في البرنامج، فقد تكون المشكلة في عملية قراءة محتوى الملف كاملًا، أو خلال معالجته وتحويله لمصفوفة الكلمات، أو خلال عملية توليد كائن إحصاء مرات التكرار للكلمات، أو قد يكون الخطأ في خوارزمية تحديد الكلمة الأكثر تكرارًا.

وأفضل أداة يمكن أن نستعين بها لتحديد الخطأ في مثل هذه الحالات هي أداة تنقيح الأخطاء، وحتى لو كانت شيفرة البرنامج الذي نعاينه قصيرة نسبيًا، فلا يفضل المرور سطرًا تلو الآخر خلال عملية التنفيذ وإضاعة الوقت، ويمكن بدلًا من ذلك الاستفادة من نقاط الوقوف للتوقف عند أماكن محددة مهمة لنا فقط، فمثلًا في نهاية جسم دالة لمعاينة القيمة التي ستعيدها.

لنبـدأ بإضـافة نقـاط وقـوف ضـمن كـل من التوابـع المسـاعدة في الملـف textHelper.js بإضـافة الكلمـة المحجوزة `debugger` ضمن الشيفرة في تلك الأماكن، لذا نفتح الملف textHelper.js ضـمن محـرر النصـوص ونضيف أول نقطة وقوف ضمن التابع `readFile()` كالتالي:

```
...

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  debugger;
  return sentences;
};

...
```

بعدها نضيف نقطة وقوف أخرى ضمن الدالة `getWords()`:

```
...

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  debugger;
  return words;
};

...
```

وأخيرًا نضيف نقطة وقوف للدالة `countWords()` كالتالي:

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (word in map) {
```

```
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  debugger;
  return map;
};

...
```

نحفظ الملف ونخرج منه، ونبدأ جلسة تنقيح الأخطاء ومع أن كل نقاط الوقوف التي أضفناها موجودة ضمن الملف textHelpers.js لكن عملية تنقيح الأخطاء ستبدأ من الملف الرئيسي للتطبيق index.js، لذا ندخل لجلسة تنقيح الأخطاء من ذلك الملف كما تعلمنا سابقًا كالتالي:

```
$ node inspect index.js
```

ليظهر لنا التالي:

```
< Debugger listening on ws://127.0.0.1:9229/b2d3ce0e-3a64-4836-bdbf-
84b6083d6d30
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in index.js:1
> 1 const textHelper = require('./textHelper');


const stopwords = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours',
'ourselves', 'you', 'your', 'yours', 'yourself', 'yourselves', 'he',
'him', 'his', 'himself', 'she', 'her', 'hers', 'herself', 'it', 'its',
'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what',
'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is',
'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had',
'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and',
'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at',
'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through',
'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up',
'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further',
'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all',
'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such',
'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too',
'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now', ''];
```

هذه المرة سننفذ الأمر c وهو اختصار للكلمة continue وتعني إكمال التنفيذ لينتقل بـذلك المنقح مباشـرة إلى أول نقطة وقوف يصل إليها تنفيذ الشيفرة، وبعد الضغط على زر الإدخال ENTER لتنفيذ الأمر يظهر التالي:

```
break in textHelper.js:6
let data = fs.readFileSync('sentences.txt');
let sentences = data.toString();
> 6   debugger;
return sentences;
};
```

نلاحظ كم من الوقت قد وفرنا في هذه العملية حيث توجهنا مباشـرة إلى أول نقطـة وقـوف، ولنتأكـد من أن هذه الدالة تعمل بشكل سليم وتقرأ محتوى الملف النصي كاملًا وتعيده، سنراقب المتغـير sentences لنعـاين قيمته ونتأكد من صحة القيمة التي تعيدها الدالة:

```
debug> watch('sentences')
```

نتقدم بالتنفيذ خطوة للأمام فقط بتنفيذ الأمر n لنعاين قيمة المتغير sentences:

```
break in textHelper.js:7
Watchers:
sentences =
    'Whale shark Rhincodon typus gigantic but harmless shark family
Rhincodontidae that is the largest living fish\n' +
      'Whale sharks are found in marine environments worldwide but
mainly in tropical oceans\n' +
      'They make up the only species of the genus Rhincodon and are
classified within the order Orectolobiformes a group containing the
carpet sharks\n' +
      'The whale shark is enormous and reportedly capable of reaching
a maximum length of about 18 metres 59 feet\n' +
      'Most specimens that have been studied however weighed about 15
tons about 14 metric tons and averaged about 12 metres 39 feet in
length\n' +
      'The body coloration is distinctive\n' +
      'Light vertical and horizontal stripes form a checkerboard
pattern on a dark background and light spots mark the fins and dark
areas of the body\n'

let sentences = data.toString();
debugger;
```

```
> 7   return sentences;
};
  9
```

تبدو القيمة صحيحة ولا مشاكل في عملية قراءة محتوى الملف إذًا فالمشكلة في مكان آخر.

لننتقل إلى نقطة الوقوف التالية بتنفيذ الأمر `c` مجددًا ليظهر ما يلي:

```
break in textHelper.js:15
Watchers:
sentences =
    ReferenceError: sentences is not defined
        at eval (eval at getWords
(your_file_path/debugger/textHelper.js:15:3), <anonymous>:1:1)
        at Object.getWords
(your_file_path/debugger/textHelper.js:15:3)
        at Object.<anonymous> (your_file_path/debugger/index.js:7:24)
        at Module._compile (internal/modules/cjs/loader.js:1125:14)
        at Object.Module._extensions..js
(internal/modules/cjs/loader.js:1167:10)
        at Module.load (internal/modules/cjs/loader.js:983:32)
        at Function.Module._load
(internal/modules/cjs/loader.js:891:14)
        at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:71:12)
        at internal/main/run_main_module.js:17:47

let words = flatSentence.split(' ');
words = words.map((word) => word.trim().toLowerCase());
>15   debugger;
return words;
};
```

رسالة الخطأ التي ظهرت سببها مراقبتنا سابقًا لقيمة المتغير `sentences` الذي لم يعد موجودًا الآن ضمن نطاق تنفيذ الدالة الحالية، حيث تبقى عملية المراقبة للمتغير طول مدة جلسة تنقيح الأخطاء، لذا سيتكرر ظهور رسالة الخطأ تلك ما دام المتغير لا يمكن الوصول إليه من مكان التنفيذ الحالي.

ويمكننا حل تلك المشكلة بإيقاف مراقبة المتغير باستخدام الدالة `unwatch()` لإيقاف مراقبة المتغير `sentences` بتنفيذ التعليمة التالية:

```
debug> unwatch('sentences')
```

لن تظهر أي رسالة عند تنفيـذ التعليمـة السـابقة، والآن لنعـود إلى الدالـة `getWords()` ونتأكـد من صـحة القيمة التي تعيدها وهي قائمة من كلمات النص السابق، لهذا نضيف مراقبة للمتغير `words` كالتالي:

```
debug> watch('words')
```

وننتقل لتنفيذ السطر التالي بتنفيذ التعليمة `n` ونعاين قيمة المتغير `words`، ونلاحظ ظهور ما يلي:

```
break in textHelper.js:16
Watchers:
words =
    [ 'whale',
      'shark',
      'rhincodon',
      'typus',
      'gigantic',
      'but',
      'harmless',
      ...
      'metres',
      '39',
      'feet',
      'in',
      'length',
      '',
      'the',
      'body',
      'coloration',
      ... ]

words = words.map((word) => word.trim().toLowerCase());
debugger;
>16   return words;
};
 18
```

لم يُظهر منقح الأخطاء محتوى المصفوفة كاملةً بسبب طولها وصعوبة قراءتها كاملة، ولكن ما ظهر يكفي ليؤكد أن محتوى النص ضمن المتغير `sentences` تم تجزئته إلى كلمات بحالة أحرف صغيرة، أي أن الدالة `getWords()` تعمل بشكل سليم.

والآن ننتقل لمعاينة الدالة الثالثة وهي `countWords()`، ولكن أولًا سنزيل المراقبة للمصفوفة `words` كي لا يظهر لنا رسالة خطأ كما حدث سابقًا عند الانتقال إلى نقطة الوقوف التالية كالتالي:

```
debug> unwatch('words')
```

ثم ننفذ الأمر `c` لينتقل التنفيذ إلى نقطة الوقوف التالية ويظهر ما يلي:

```
break in textHelper.js:29
});

>29   debugger;
return map;
};
```

سنتأكد ضمن هذه الدالة من احتواء المتغير `map` على كل الكلمات السابقة مع قيم تكرارها، لذا نبدأ مراقبة المتغير `map` كالتالي:

```
debug> watch('map')
```

ثم ننتقل بالتنفيذ إلى السطر التالي بتنفيذ الأمر `n` ليظهر لنا ما يلي:

```
break in textHelper.js:30
Watchers:
map =
    { 12: NaN,
NaN,
NaN,
NaN,
NaN,
NaN,
      whale: 1,
      shark: 1,
      rhincodon: 1,
      typus: NaN,
      gigantic: NaN,
```

```
      ... }

 28
debugger;
>30   return map;
};
 32
```

على ما يبدو أن هذه الدالة هي سبب المشكلة وعملية إحصاء تكرار الكلمات خاطئة، ولمعرفة سبب الخطأ يجب أن نعاين عمل هذه الدالة ضمن حلقة المرور على عناصر المصفوفة `words`، لذا سنعدل أماكن نقاط الوقوف الحالية.

نبدأ بالخروج من منقح الأخطاء بتنفيذ الأمر التالي:

```
debug> .exit
```

ثم نفتح الملف textHelper.js ضمن محرر النصوص لنعدل نقاط الوقوف ضمنه:

```
$ nano textHelper.js
```

بما أننا تأكدنا من صحة عمل الدالتين `readFile()` و `getWords()` سنزيل نقاط الوقوف من داخلهما، ونزيل نقطة الوقوف من نهاية الدالة `countWords()` ونضيف نقطتي وقوف جديدتين في بداية ونهاية الدالة `forEach()`، ليصبح الملف textHelper.js كالتالي:

```
...

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  return sentences;
};

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  return words;
```

```
};

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    debugger;
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
    debugger;
  });

  return map;
};

...
```

نحفظ الملف ونخرج منه ثم نبدأ جلسة تنقيح أخطاء جديدة كالتالي:

```
$ node inspect index.js
```

كي نحدد سبب المشكلة يجب أن نراقب عدة قيم، أولها قيمة الكلمة الحالية `word` المُمررة كمعامل من قبل تابع حلقة التكرار `forEach()` كالتالي:

```
debug> watch('word')
```

لا تقتصر ميزة المراقبة ضمن جلسة تنقيح الأخطاء على المتغيرات فحسب، بل يمكن مراقبة قيم تعابير جافاسكربت البرمجية المستخدمة ضمن الشيفرة، كأن نراقب قيمة تنفيذ التعليمة الشرطية `word in map` والتي تحدد ما إذا كانت الكلمة الحالية موجودة مسبقًا، ويمكن مراقبتها بتنفيذ التالي:

```
debug> watch('word in map')
```

لنضيف مراقبة لقيمة تكرار الكلمة الحالية ضمن متغير النتيجة `map` كالتالي:

```
debug> watch('map[word]')
```

لا تقتصر ميزة المراقبة على التعابير البرمجية الموجودة ضمن الشيفرة فحسب، بل يمكن إضافة أي تعابير برمجية نريدها ليتم تنفيذها ومراقبة قيمتها، لذا سنستفيد من هذه الميزة ونضيف مراقبة لقيمة طول الكلمة الحالية ضمن المتغير `word`:

```
debug> watch('word.length')
```

بعد أن انتهينا من إضافة القيم التي نريد مراقبتها أثناء التنفيذ سننفذ الأمر `c` ونراقب كيف تعالج الدالة أول كلمة من مصفوفة الكلمات ضمن الحلقة داخل الدالة `countWords()`، ليظهر لنا ما يلي:

```
break in textHelper.js:20
Watchers:
word = 'whale'
word in map = false
map[word] = undefined
word.length = 5

let map = {};
words.forEach((word) => {
>20     debugger;
if (word in map) {
map[word] = 1;
```

الكلمة الأولى التي يتم معالجتها هي `whale` ولا يحوي الكائن `map` على مفتاح للكلمة `whale` لأنه فارغ، لذا قيمة المراقبة للكلمة الحالية `whale` ضمن الكائن `map` كما نلاحظ هي `undefined`، وطول الكلمة الحالية `whale` هو `5`، وهذه القيمة تحديدًا لا تفيدنا في البحث عن سبب الخطأ، ولكننا أضفناها لنتعلم كيف يمكن حساب ومراقبة أي تعبير برمجي خلال جلسة تنقيح الأخطاء.

والآن ننفذ التعليمة `c` لنرى ماذا سيحدث في نهاية تنفيذ الدورة الحالية ليظهر لنا ما يلي:

```
break in textHelper.js:26
Watchers:
word = 'whale'
word in map = true
map[word] = NaN
word.length = 5

map[word] += 1;
}
```

```
>26     debugger;
});
 28
```

أصبحت قيمة العبارة `word in map` صحيحة `true` بسبب إضافة مفتاح للكلمة الحالية `whale` ضمن الكائن `map`، ولكن قيمة المفتاح `whale` ضمن الكائن `map` هي `NaN` ما يدل على وجود مشكلة ما، وتحديدًا في العبارة الشرطية `if` ضمن الدالة `countWords()`، فوظيفتها هي تحديد فيما إذا كنا سنضيف مفتاحًا جديدًا للكلمة الحالية إذا لم تكن موجودة سابقًا، أو إضافة واحد لقيمة المفتاح إن كان موجودًا مسبقًا، والصحيح هو تعيين القيمة `map[word]` إلى `1` إذا لم تكن الكلمة `word` موجودة كمفتاح ضمن `map`، بينما حاليًا نحن نضيف قيمة واحد في حال العثور على `word` وهو عكس المطلوب.

وكما لاحظنا في بداية الحلقة كانت قيمة التكرار للكلمة الحالية `map["whale"]` غير موجودة `undefined`، وفي جافاسكربت إذا حاولنا إضافة واحد إلى تلك القيمة `undefined + 1` سينتج عن تلك العملية القيمة `NaN` وهو ما ظهر بالفعل، ولتصحيح هذه المشكلة يمكننا تعديل الشرط ضمن `if`، فبدلًا من أن يكون `word in map` ننفي هذه العبارة لتصبح كالتالي `!(word in map)`، حيث يُستخدم الرمز `!` لنفي العبارات المنطقية فيصبح الشرط صحيحًا إذا لم يحتوي الكائن `map` على مفتاح للقيمة `word`.

والآن لننفذ هذا التعديل ضمن الدالة `countWords()` ونختبرها مجددًا، لكن نخرج أولًا من جلسة تنقيح الأخطاء كالتالي:

```
debug> .exit
```

ونفتح الملف textHelper.js مجددًا ضمن محرر النصوص:

```
$ nano textHelper.js
```

نعدل الدالة `countWords()` بالشكل التالي:

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (!(word in map)) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
```

```
  });

  return map;
};

...
```

نحفظ الملف ونخرج منه، وننفذ البرنامج ونراقب النتيجة:

```
$ node index.js
```

تظهر لنا النتيجة التالية هذه المرة:

```
The most popular word in the text is "whale" with 3 occurrences
```

وهي إجابة منطقية وأفضل من السابقة، ونلاحظ كيف ساعدنا منقح الأخطاء في تحديد الدالة التي كانت سبب المشكلة وتمييز الدوال التي تعمل بشكل سليم وساعدنا في اكتشاف سبب الخطأ، وبذلك نكون قد تعلمنا طريقة استخدام منقح الأخطاء الخاص بنود من سطر الأوامر.

وتعلمنا أيضًا كيف يمكن إضافة نقاط الوقوف باستخدام الكلمة `debugger` وإعداد مراقبة لمختلف القيم والعبارات البرمجية لمراقبة حالة البرنامج أثناء التنفيذ وكل ذلك من سطر الأوامر، ولكن لتوفير تجربة استخدام أسهل يمكن إجراء العملية نفسها عبر واجهة مستخدم مرئية، وهذا ما سنتعرف عليه في الفقرة التالية.

سنتعلم في الفقرة التالية طريقة استخدام منقح الأخطاء من أدوات المطور في متصفح جوجل كروم، حيث سنبدأ جلسة لتنقيح الأخطاء في نود كما فعلنا سابقًا، وسنستعمل صفحة مخصصة من واجهة متصفح كروم لتعيين نقاط الوقوف وعمليات المراقبة من واجهة مرئية بدلًا من سطر الأوامر.

10.3 تنقيح الأخطاء في نود باستخدام أدوات المطور في كروم

تعد أدوات المطور في متصفح كروم من أشهر أدوات تنقيح الأخطاء لشيفرة جافاسكربت عمومًا ونود خصوصًا ضمن متصفح الويب، وذلك لأن محرك جافاسكربت المستخدم من قبل نود V8 هو نفسه المستخدم في متصفح كروم، لذا فالتكامل بينهما يوفر تجربة مرنة لتنقيح الأخطاء.

سنطبق في هذه الفقرة على مثال بسيط وهو خادم HTTP في نود مهمته إعادة قيمة بصيغة JSON كرد على الطلبات الواردة، وسنستخدم لاحقًا منقح الأخطاء لإعداد نقاط الوقوف ومراقبة عمل ذلك الخادم وتحديدًا كيف يتم توليد قيمة الرد على الطلبات الواردة، وللمزيد حول عملية إنشاء الخادم، راجع مقالة إنشاء خادم ويب في Node.js باستخدام الوحدة HTTP.

نبدأ بإنشاء ملف جافاسكربت جديد بالاسم server.js سيحوي على برنامج الخادم ونفتح الملف ضمن محرر النصوص كالتالي:

```
$ nano server.js
```

مهمة الخادم هي إعادة العبارة `Hello World` بصيغة JSON ضمن الرد، حيث سيحوي على مصفوفة لعدة ترجمات لتلك العبارة ليختار إحداها عشوائيًا ويعيدها ضمن جسم الرد بصيغة JSON، وسيستمع الخادم إلى الطلبات الواردة على العنوان المحلي `localhost` وعلى المنفذ رقم `8000:`.

والآن نبدأ بإضافة شيفرة البرنامج كما يلي:

```
const http = require("http");

const host = 'localhost';
const port = 8000;

const greetings = ["Hello world", "Hola mundo", "Bonjour le monde",
"Hallo Welt", "Salve mundi"];

const getGreeting = function () {
  let greeting = greetings[Math.floor(Math.random() *
greetings.length)];
  return greeting
}
```

استوردنا الوحدة برمجية `http` والتي تساعد في إعداد خادم HTTP، ثم وضعنا قيم عنوان الخادم ورقم المنفذ ضمن المتغيرين `host` و `port` لاستخدامها لاحقًا لتشغيل الخادم، ثم عرفنا مصفوفة العبارات `greetings` والتي تحوي على جميع العبارات الممكن إرسالها من قبل الخادم لتختار الدالة `getGreeting()` إحداها عشوائيًا ويعيده.

والآن سنضيف دالة معالجة طلبات HTTP القادمة للخادم وشيفرة بدء تشغيل الخادم كالتالي:

```
...

const requestListener = function (req, res) {
  let message = getGreeting();
  res.setHeader("Content-Type", "application/json");
  res.writeHead(200);
  res.end(`{"message": "${message}"}`);
```

```
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
  console.log(`Server is running on http://${host}:${port}`);
});
```

أصبح الخادم بذلك جاهزًا للخطوة التالية وهي إعداد منقح أخطاء كروم، لهذا نبدأ جلسة تنقيح الأخطاء بتنفيذ الأمر التالي:

```
$ node --inspect server.js
```

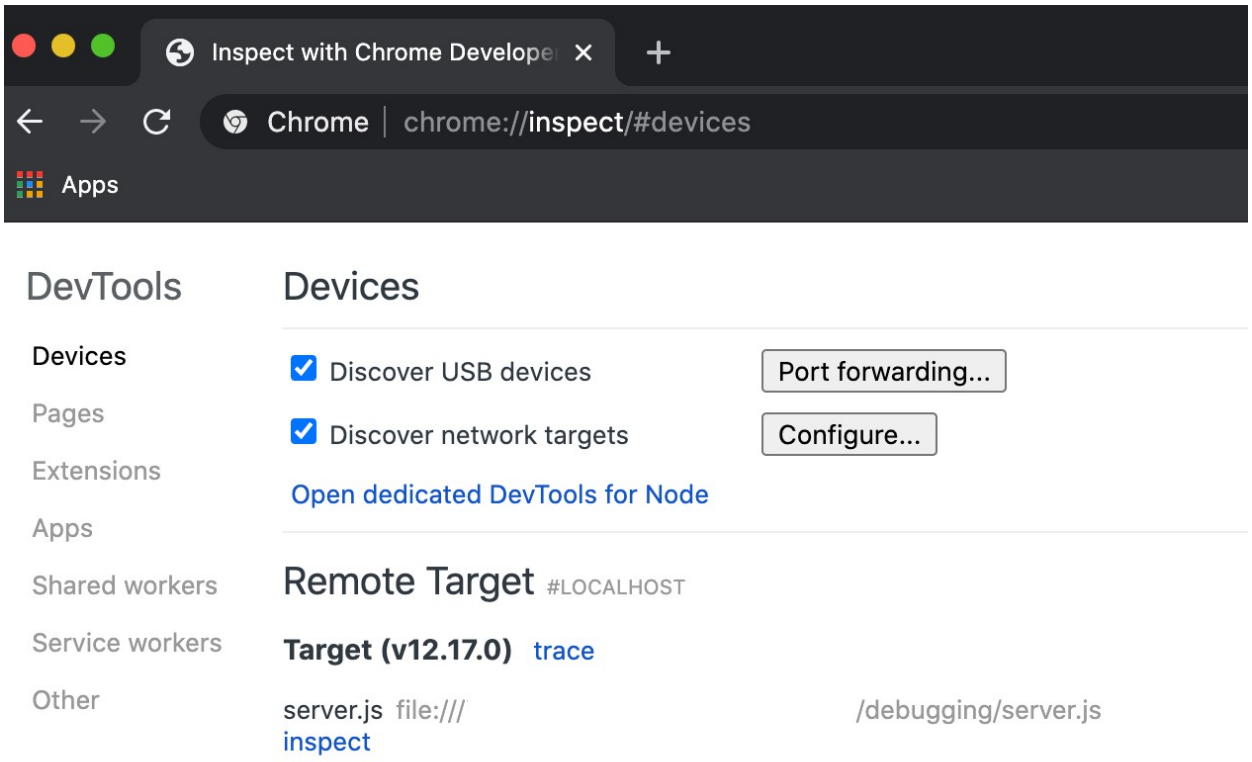
> نلاحظ الفرق بين أمر بدء منقح الأخطاء الخاص بنود من سطر الأوامر وبين أمر منقح الأخطاء الخاص بكروم، حيث ننفذ الأمر `inspect` للأول، أما للثاني نمرر الخيار `--inspect`.

وبعد تشغيل منقح الأخطاء سنلاحظ ظهور ما يلي:

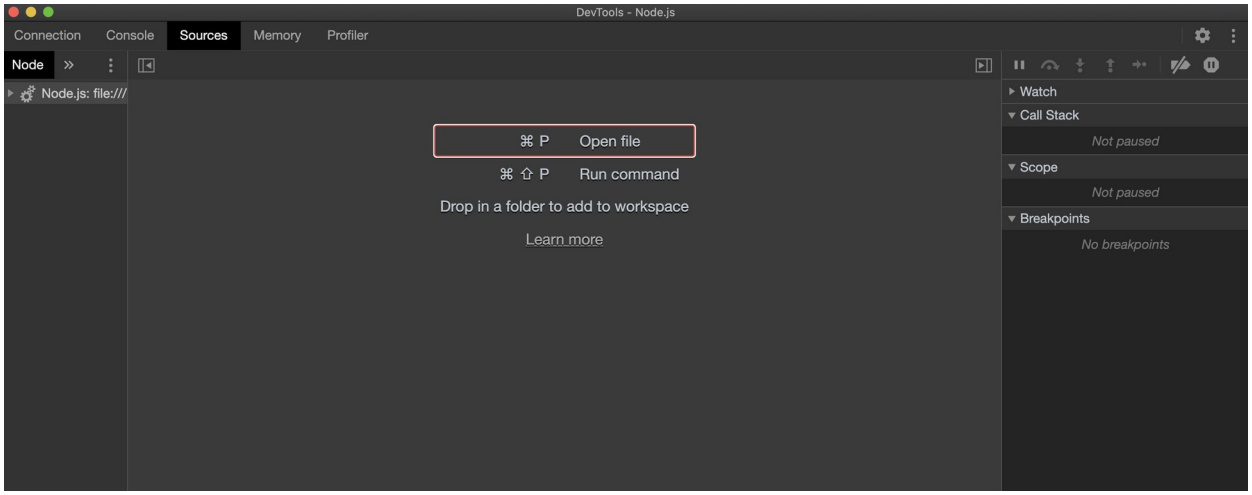
```
Debugger listening on ws://127.0.0.1:9229/996cfbaf-78ca-4ebd-9fd5-893888efe8b3
For help, see: https://nodejs.org/en/docs/inspector
Server is running on http://localhost:8000
```

يمكننا الآن فتح متصفح جوجل كروم أو كروميوم Chromium والـذهاب للعنـوان chrome://inspect من شريط العنوان في الأعلى، ويمكن أيضًا استعمال منقح الأخطاء لمتصفح مايكروسوفت إيدج Microsoft Edge ولكن بالذهاب إلى العنوان edge://inspect بدلًا من العنوان السابق.

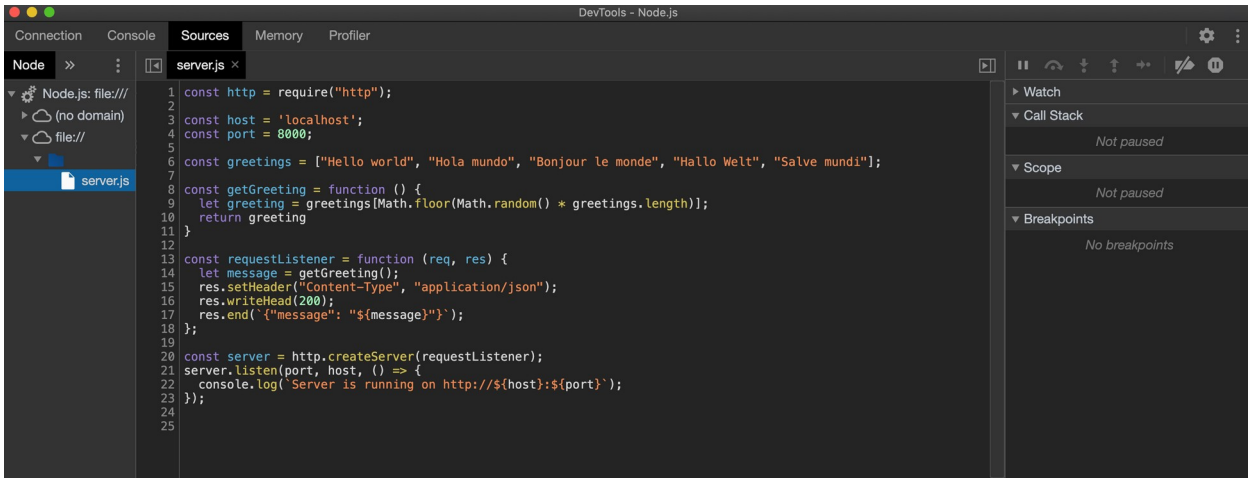
وبعد الذهاب لذلك العنوان ستظهر لنا الصفحة التالية:



نـــذهب لقســـم الأجهـــزة Devices ونضـــغط على أمـــر فتح أداوت المطـــور الخاصـــة بنـــود "Open dedicated DevTools for Node" لتظهر لنا نافذة منفصلة كالتالي:



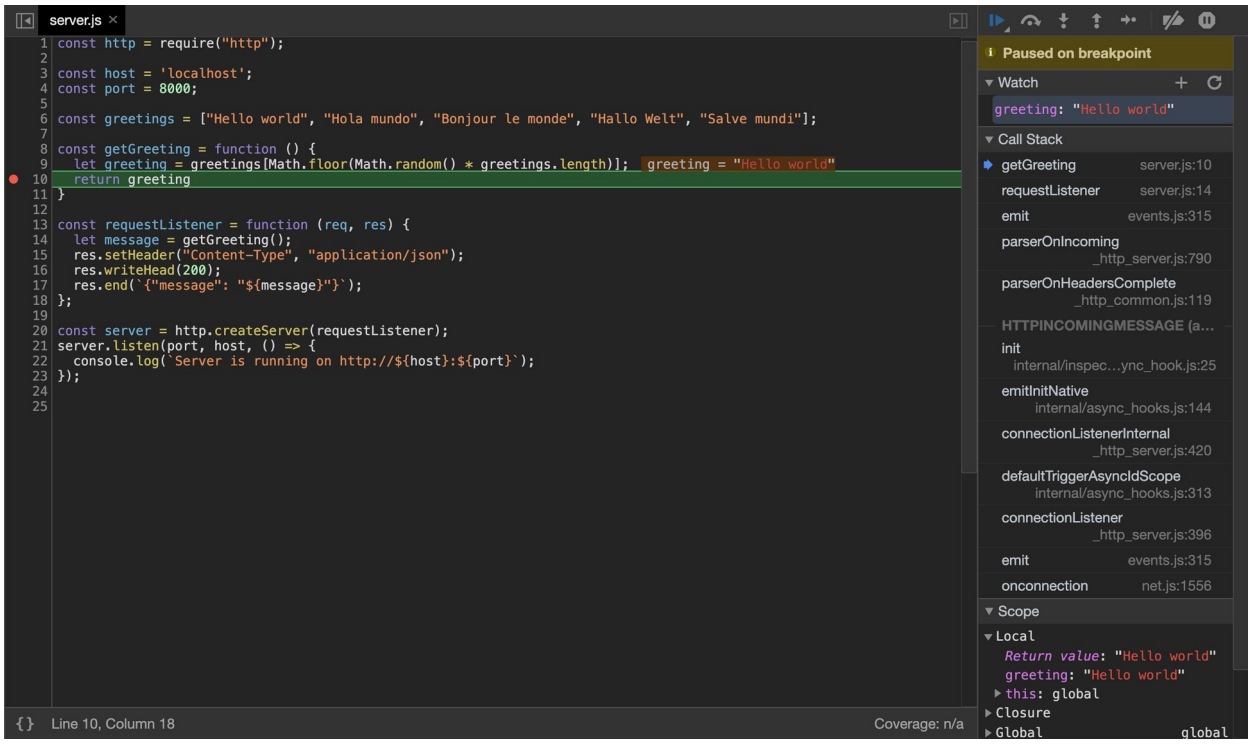
يمكننا الآن تنقيح أخطاء برنامج نود الســابق بواســطة كــروم، لــذلك نــذهب إلى تبــويب المصــادر Sources ونوسع قسم شجرة الملفات الظاهر على اليسار ونختار منه ملف البرنامج الخاص بنا وهو server.js:



ونضيف نقطة وقوف ضمن الشيفرة التي تظهر، حيث نريد التوقف بعد أن يختار البرنامج عبارة الترحيب التي سيعيدها ضمن الرد لنعاينها، لذلك يمكننا الضغط مباشرة على رقم السطر **10** لتظهر نقطة حمراء بجانبه ما يدل على إضافة نقطة وقوف في هذا السطر، وهو ما نلاحظه من قائمة نقاط الوقوف في اللوحة على اليمين:

لنراقب الآن عبارة برمجية، حيث يمكننا ذلك من اللوحة على اليمين وتحديدًا بجانب عنوان قسم المراقبة Watch بالضغط على علامة الزائد "+"، ونضيف اسم المتغير `greeting` لنراقب قيمته أثناء التنفيذ ثم نضغط على زر الإدخال ENTER.

والآن لنبدأ بتنقيح البرنامج، فنذهب ضمن نافذة المتصفح إلى عنوان الذي يستمع إليه الخادم `http://localhost:8000` وبعد الضغط على زر الإدخال ENTER للذهاب إلى ذلك العنوان سنلاحظ عدم ظهور أي رد مباشرة بل ستظهر لنا نافذة تنقيح الأخطاء في الواجهة مجددًا، وفي حال لم تظهر النافذة يمكن الذهاب إليها يدويًا لنلاحظ ظهور ما يلي:



حيث توقف تنفيذ الخادم عند نقطة الوقوف التي عيّناها سابقًا، ونلاحظ تحديث قيم المتغيرات التي نراقبها في لوحة المراقبة على الجانب الأيمن، وكذلك تظهر تلك القيمة بجانب السطر الحالي ضمن الشيفرة.

ولمتابعة تنفيذ الشيفرة نضغط على زر المتابعة الموجود في اللوحة على الجانب الأيمن فوق العبارة "Paused on breakpoint" والتي تعني توقف التنفيذ عند نقطة الوقوف، وبعد اكتمال التنفيذ ستلاحظ ظهور رد بصيغة JSON ضمن نافذة المتصفح التي تواصلنا منها مع الخادم:

```
{"message": "Hello world"}
```

نلاحظ أننا لم نضيف أي عبارات ضمن الشيفرة أو نعدل عليها لإضافة نقاط الوقوف، وهي الفائدة التي تقدمها أدوات تنقيح الأخطاء من الواجهة المرئية مثل كروم، وهو الخيار الأفضل لمن لا يرغب بالتعامل مع سطر الأوامر ويفضل التعامل مع الواجهات المرئية.

10.4 خاتمة

تعلمنا في هذا الفصل طريقة التعامل مع منقح الأخطاء في تطبيقات نود وطريقة إعداد الراصدات لمراقبة حالة التطبيق، وتعلمنا طريقة استخدام نقاط الوقوف لمعاينة تنفيذ البرنامج في عدة أماكن ضمن البرنامج أثناء عمله، وتعاملنا مع كل من منقح أخطاء نود من سطر الأوامر ومن متصفح جوجل كروم من أدوات المطور الخاصة به، وذلك بدلًا من إضافة تعليمات الطباعة للقيم المختلفة داخل البرنامج.

يمكن الاستعانة بمنقح الأخطاء ما يسهل عملية استكشاف أخطاء التنفيذ ضمن البرنامج ومعاينة حالتـه، مـا يوفر من وقت التطوير وخصوصًا وقت حل المشكلات وإصلاح الأخطاء.

ويمكن الرجوع إلى توثيق نود الرسـمي عن أدوات تنقيح الأخطـاء أو دليـل أدوات المطـور من كـروم ودليـل أدوات المطور لتنقيح شيفرة جافاسكربت.

11. التعامل مع العمليات الأبناء

Child Process

عند تشغيل أي برنامج في نود Node.js ستعمل نسخة منه افتراضيًا ضمن عملية process واحدة في نظام التشغيل، وسيُنفذ فيها البرنامج ضمن خيط معالجة thread وحيد، وكما تعلمنا في الفصل الخامس فإن تنفيذ البرنامج ضمن خيط وحيد ضمن العملية سيؤدي لأن تعيق العمليات التي تحتاج مدة طويلة لتنفيذها في جافاسكربت تنفيذ العمليات أو الشيفرات التي تليها ضمن خيط التنفيذ لحين انتهاءها، وهنا يأتي دور إنشاء عملية ابن child process منفصلة عن العملية الرئيسية، وهي عملية تُنشئها عملية أخرى وتُستخدم لتنفيذ المهام الطويلة، وبهذه الطريقة يمكن لنظام التشغيل تنفيذ كلا العمليتين الأب والابن معًا أو بنفس الوقت على التوازي دون أن يعيق أي منهما تنفيذ الآخر.

توفر نود لذلك الغرض الوحدة البرمجية child_process التي تحتوي على توابع عدة تساعد في إنشاء عمليات جديدة، وحتى توابع للتعامل مع نظام التشغيل مباشرةً وتنفيذ الأوامر ضمن الصدفة shell، لذا يمكن لمسؤولي إدارة النظام الاستفادة من نود في تنفيذ أوامر الصدفة لإدارة نظام التشغيل وترتيب تلك الأوامر ضمن وحدات برمجية بدلًا من تنفيذ ملفات أوامر الصدفة مباشرةً.

سنتعلم في هذا الفصل طرق إنشاء عمليات أبناء بتطبيق عدة أمثلة حيث سننشئ تلك العمليات بالاستعانة بالوحدة البرمجية `child_process` ونعاين نتيجة تنفيذها على شكل مخزن مؤقت buffer أو سلسلة نصية باستخدام التابع `exec()`، وسنتعلم كيف يمكن قراءة نتيجة تنفيذ تلك العملية من مجرى للبيانات data stream باستخدام التابع `spawn()`، ثم سننفذ برنامج نود آخر ضمن عملية منفصلة باستخدام `fork()` ونتعلم طريقة التواصل معه أثناء تشغيله، وسنطبق هذه الأفكار على مثال لبرنامج مهمته عرض قائمة محتويات مجلد ما، وبرنامج آخر للبحث عن الملفات، وآخر لخادم ويب يدعم عدة مسارات فرعية.

11.1 إنشاء عملية ابن باستخدام exec

عادة ما ننشئ عملية ابن لتنفيذ بعض الأوامر ضمن نظام التشغيل، فمثلًا للتعديل على خرج برنامج ما في نود بعد تنفيذه ضمن الصدفة نمرر خرج ذلك البرنامج أو نعيد توجيهه إلى أمر آخر، وهنا يأتي دور التابع exec() الذي يمكننا من إنشاء عملية صدفة جديدة بنفس الطريقة وتنفيذ الأوامر ضمنها لكن من قبل برنامج نود، حيث يُخزَّن خرج ذلك الأمر ضمن مخزن مؤقت في الذاكرة ويمكننا بعدها الوصول إليه بتمرير دالة رد نداء callback function للتابع `exec()`.

لنبدأ بإنشاء عملية ابن جديدة في نود ولكن أولًا ننشئ مجلد جديد سيحتوي على البرامج التي سنعمل عليها في هذا الفصل بالاسم `child-processes` كالتالي:

```
$ mkdir child-processes
```

وندخل إلى المجلد:

```
$ cd child-processes
```

ننشئ ملف جافاسكربت جديد بالاسم listFiles.js ونفتحه باستخدام أي محرر نصوص:

```
$ nano listFiles.js
```

سنستخدم في هذه الوحدة البرمجية التابع `exec()` لتنفيذ أمر عرض الملفات والمجلدات ضمن المجلد الحالي `ls`، ومهمة برنامجنا هو قراءة خرج ذلك الأمر وعرضه للمستخدم، لذا نضيف الشيفرة التالية:

```
const { exec } = require('child_process');

exec('ls -lh', (error, stdout, stderr) => {
  if (error) {
    console.error(`error: ${error.message}`);
    return;
  }

  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }

  console.log(`stdout:\n${stdout}`);
```

```
});
```

بدأنا باستيراد التابع `exec()` من الوحدة `child_process`، ثم استدعيناه بتمرير الأمر الـذي نريـد تنفيـذه كمعامل أول، وهو الأمر `ls -lh` الذي سيعرض كافـة الملفـات والمجلـدات الموجـودة ضـمن المجلـد الحـالي بصيغة مفصلة، وسيعرض وحدة الحجم للملفات بصيغة مقروءة، وسيعرض أيضًا الحجم الكلي لها في أول سطر من الخرج.

والمعامل الثاني المُمرر هو دالة رد النداء تقبـل ثلاث معـاملات، الأول كـائن الخطـأ `error` والثـاني الخـرج القياسي `stdout` والثالث خرج الخطأ `stderr`، فإذا فشل تنفيـذ الأمـر سـيحتوي المعامـل `error` على كـائن خطأ يشرح سبب حدوثه مثلًا عندما لا تعثر الصدفة على الأمر الذي نحاول تنفيذه، وإذا نُفذ الأمر بنجاح سـيحتوي المعامل الثاني `stdout` على البيانات التي تكتب في مجـرى الخـرج القياسـي، أمـا المعامـل الثـالث `stderr` سيمثل مجرى الخطأ القياسي ويحوي على أي بيانات يكتبها الأمر إلى ذلك المجرى.

> يوجد فرق بين كائن الخطأ `error` ومجرى الخطأ `stderr`، فإذا فشل تنفيذ الأمر كليًا سيمثل المعامل `error` ذلك الخطأ، بينما إذا نُفذ الأمر وكتب هو إلى مجرى الخطأ فيمكننا قراءة أي بيانات تكتب فيه من المعامل `stderr`، ويفضل دومًا معالجة كل احتمالات الخرج الممكنة من كلا هذين المعاملين مع أي عملية ابن.

نتحقق داخل دالة رد النداء الممررة من وجود أي خطأ أولًا، فإذا وُجد خطـأ سـنطبع رسـالة الخطـأ `message` وهي الخاصية ضمن كائن الخطأ `Error` باستدعاء أمر طباعة الخطأ `console.error()`، ثم ننهي تنفيذ التابع مباشرةً باستخدام `return`، وبعدها نتحقق من طباعة الأمر لأي أخطاء تُكتَب ضمن مجرى الخطـأ القياسـي وإذا وجد نطبع الرسالة وننهي تنفيذ التابع باستخدام `return` أيضًـا، وإلا يكـون الأمـر قـد نُفِّذ بنجـاح، ونطبـع حينهـا الخرج إلى الطرفية باستخدام `console.log()`.

والآن نخرج من الملف ثم ننفذ البرنامج ونعاين النتيجة، وفي حال كنت تستخدم محرر النصوص نانو nano كما في أمثلتنـا يمكنـك الخـروج منـه بالضـغط على الاختصـار CTRL+X، ولتشـغيل البرنـامج ننفـذ الأمـر `node` كالتالي:

```
$ node listFiles.js
```

نحصل على الخرج:

```
stdout:
total 4.0K
-rw-rw-r-- 1 hassan hassan 280 Jul 27 16:35 listFiles.js
```

وهو محتوى المجلد child-processes مع تفاصيل عن الملفات الموجودة ضمنه، وحجم المجلد الكلي في السطر الأول، وهو ما يدل على تنفيذ البرنامج listFiles.js للأمر `ls -lh` ضمن الصدفة وقراءة نتيجته وطباعتها بنجاح.

والآن سنتعرف على طريقة مختلفة لتنفيذ عملية ما على التوازي مع العملية الحالية، حيث توفر الوحدة `child_process` التابع `execFile()` الذي يُمكننا من تشغيل الملفات التنفيذية، والفرق بينه وبين الأمر `exec()` أن المعامل الأول المُمرر له سيكون مسار الملف التنفيذي الذي نريد تشغيله بدلًا من أمر يراد تنفيذه في الصدفة، وبطريقة مشابهة لعمل التابع `exec()` سيُخزن ناتج التنفيذ ضمن مخزن مؤقت يمكننا الوصول إليه ضمن دالة رد النداء الممررة، والتي تقبل المعاملات الثلاث نفسها `error` و `stdout` و `stderr`.

> يجب الانتباه أنه لا يمكن تشغيل الملفات التنفيذية ذات الصيغ `bat.` و `cmd.` على ويندوز، وذلك لأن التابع `execFile()` لا ينشئ الصدفة التي تحتاج إليها تلك الملفات لتشغيلها، بينما على الأنظمة مثل يونكس ولينكس و نظام ماك لا تحتاج الملفات التنفيذية إلى صدفة لتشغيلها، لذا لتنفيذ الملفات التنفيذية على ويندوز يمكن استخدام التابع `exec()` لأنه سيُنشئ لها صدفة عند التنفيذ، أو يمكن استدعاؤها باستخدام التابع `spawn()` وهو ما سنتعرف عليه لاحقًا، ولكن الملفات التنفيذية ذات اللاحقة `exe.` يمكن تشغيلها ضمن ويندوز باستخدام `execFile()` مباشرةً، حيث أنها لا تحتاج لصدفة لتشغيلها.

والآن نبدأ بإنشاء الملف التنفيذي الذي سنحاول تنفيذه باستخدام `execFile()`، حيث سنكتب نصًا برمجيًا ضمن صدفة باش bash مهمته تنزيل صورة شعار بيئة نود من الموقع الرئيسي لها، ثم يعيد ترميز صورة الشعار تلك بصيغة Base64 لنتعامل معها كسلسلة نصية بمحارف ASCII، ونبدأ بإنشاء ملف تنفيذي جديد بالاسم processNodejsImage.sh:

```
$ nano processNodejsImage.sh
```

ونضيف إليه الشيفرة التالية لتحميل وتحويل صورة الشعار:

```
#!/bin/bash
curl -s https://nodejs.org/static/images/logos/nodejs-new-pantone-
black.svg > nodejs-logo.svg
base64 nodejs-logo.svg
```

التعليمة في السطر الأول تسمى شِبانغ shebang، وتستخدم ضمن أنظمة يونكس ولينكس ونظام ماك لتحديد الصدفة التي نريد تشغيل النص البرمجي أو السكربت ضمنها، والتعليمة التالية هي الأمر curl وهي أداة سطر أوامر تمكننا من نقل البيانات من وإلى الخوادم، ويمكننا الاستفادة منها لتنزيل شعار نود من الموقع الرئيسي له، ثم نعيد توجيه الخرج لحفظ الصورة بعد تنزيلها إلى ملف بالاسم nodejs-logo.svg، أما التعليمة الأخيرة تستخدم الأداة `base64` لإعادة ترميز محتوى ملف الشعار nodejs-logo.svg الذي نزلناه سابقًا، ثم سيُطبَع نتيجة الترميز إلى الطرفية أي مجرى الخرج القياسي وهو خرج تنفيذ النص البرمجي هذا بالكامل.

والآن نحفظ الملف ونخرج منه ونضيف إذن تنفيذ هذا النص البرمجي لكي نستطيع تنفيذه كالتالي:

```
$ chmod u+x processNodejsImage.sh
```

يمنح هذا الأمر المستخدم الحالي صلاحية التنفيذ لذلك الملف.

يمكننا الآن البدء بكتابة برنامج نود الذي سيُنفذ ذلك النص البرمجي باستخدام التـابع `execFile()` ضـمن عملية ابن منفصلة ثم طباعة خرج التنفيذ، لذا نُنشئ ملف جافاسكربت جديد بالاسم getNodejsImage.js:

```
$ nano getNodejsImage.js
```

ونكتب الشيفرة التالية:

```
const { execFile } = require('child_process');

execFile(__dirname + '/processNodejsImage.sh', (error, stdout, stderr)
=> {
  if (error) {
    console.error(`error: ${error.message}`);
    return;
  }

  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }

  console.log(`stdout:\n${stdout}`);
});
```

اسـتوردنا التـابع `execFile()` من الوحـدة `child_process` واسـتدعيناه بتمريـر مسـار ملـف النص البرمجي، حيث استفدنا من قيمة الثابت `__dirname` الذي توفره نود للحصول على مسار المجلد الحالي الـذي يحتوي على النص الـبرمجي، وبـذلك يمكن للبرنـامج الإشـارة إلى النص الـبرمجي processNodejsImage.sh دومًا مهما كان نظام التشغيل الذي ينفذه أو مكان تنفيـذ البرنـامج getNodejsImage.js على نظـام الملفـات، وفي حالتنـا يجب أن يكـون مكـان كـل من الملفين getNodejsImage.js و processNodejsImage.sh في نفس المجلد.

أما المعامل الثاني المُمرر هو رد نداء ويقبل ثلاثة معاملات، الأول كائن الخطأ `error` والثاني الخرج القياسي `stdout` والثالث خرج الخطأ `stderr`، وكما فعلنا سابقًا عند استخدام `exec()` سنتحقق من حالة وخرج التنفيذ ونطبعها إلى الطرفية.

والآن نحفظ الملف ونخرج من محرر النصوص ثم نشغله باستخدام الأمر `node` كالتالي:

```
$ node getNodejsImage.js
```

لنحصل على الخرج:

```
stdout:
PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHhtbG5zOnhsaW5rPS
JodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hsaW5rIiB2aWV3Qm94PSIwIDAgNDQyLjQgMjcw
LjkiPjxkZWZzPjxsaW5lYXJHcmFkaWVudCBpZD0iYiIgeDE9IjE4MC43IiB5MT0iODAuNy
Ige
...
```

تجاهلنا عرض الخرج كاملًا بسبب حجمه الكبير، ولكن النص البرمجي processNodejsImage.sh نزل الصورة أولًا بعدها أعاد ترميزها بصيغة base64، ويمكن التأكد من ذلك بمعاينة الصورة التي تم تنزيلها والموجودة ضمن المجلد الحالي، ولنتأكد يمكننا تنفيذ البرنامج السابق listFiles.js لمعاينة المحتوى الجديد للمجلد الحالي:

```
$ node listFiles.js
```

سنلاحظ ظهور الخرج التالي:

```
stdout:
total 20K
-rw-rw-r-- 1 hassan hassan  316 Jul 27 17:56 getNodejsImage.js
-rw-rw-r-- 1 hassan hassan  280 Jul 27 16:35 listFiles.js
-rw-rw-r-- 1 hassan hassan 5.4K Jul 27 18:01 nodejs-logo.svg
-rwxrw-r-- 1 hassan hassan  129 Jul 27 17:56 processNodejsImage.sh
```

بذلك نكون قد نفذنا بنجاح النص البرمجي processNodejsImage.sh ضمن عملية ابن من برنامج نود باستخدام التابع `execFile()`.

تعلمنا في هذه الفقرة كيف يمكن للتابعين `exec()` و `execFile()` تنفيذ الأوامر ضمن صدفة نظام التشغيل داخل عملية ابن منفصلة في نود، وتوفر نود أيضًا التابع `spawn()` والذي يشبه في عمله هذين التابعين، ولكن الفرق في عمله أنه لا يقرأ خرج تنفيذ الأمر دفعة واحدة بل على عدة دفعات ضمن مجرى للبيانات stream، وهو ما سنتعرف عليه بالتفصيل في الفقرة التالية.

11.2 إنشاء عملية ابن باستخدام spawn

يمكن استدعاء التابع spawn() لتنفيذ الأوامر ضمن عملية منفصلة والحصول على بيانات الخرج من ذلك الأمر عن طريق الواجهة البرمجية API لمجرى البيانات في نود، وذلك عبر الاستماع لبعض الأحداث المعينة على كائن المجرى لخرج ذلك الأمر.

مجاري البيانات streams في نود هي نسخة من صنف مرسل الأحداث event emitter الذي تعرفنا عليه بالتفصيل في الفصل التاسع من هذه السلسلة وعندما يكون خرج الأمر الذي سننفذه كبير نسبيًا فيفضل استخدام التابع spawn() بدلًا من التابعين exec() و execFile()، وذلك لأن التابعين exec() و execFile() سيخزنان خرج الأمر كاملًا ضمن مخزن مؤقت في الذاكرة، ما سيؤثر على أداء النظام، بينما باستعمال المجرى stream يمكننا قراءة البيانات من الخرج ومعالجتها على عدة دفعات، ما يؤدي لخفض استعمال الذاكرة والسماح لنا بمعالجة البيانات الكبيرة.

سنتعرف الآن على طريقة استخدام التابع spawn() لإنشاء عملية ابن، لذلك نبدأ بكتابة برنامج في نود مهمته تنفيذ أمر البحث عن الملفات find ضمن عملية ابن لعرض كل الملفات الموجودة ضمن المجلد الحالي، ونبدأ بإنشاء ملف جافاسكربت جديد بالاسم findFiles.js:

```
$ nano findFiles.js
```

ونستدعي التابع spawn() لتنفيذ أمر البحث:

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);
```

بدأنا باستيراد التابع spawn() من الوحدة child_process، ثم استدعيناه لإنشاء عملية ابن جديدة يُنفذ ضمنها الأمر find وخزنّا نتيجة تنفيذ التابع ضمن المتغير child للاستماع لاحقًا إلى الأحداث الذي ستطلقها العملية الابن، ونلاحظ تمرير الأمر الذي نريد تنفيذه find كمعامل أول للتابع spawn()، أما المعامل الثاني فهو مصفوفة من المعاملات التي نريد تمريرها لذلك الأمر، ويكون الأمر النهائي الذي سينفذ هو أمر البحث find مع تمرير المعامل . للدلالة على البحث عن كل الملفات الموجودة ضمن المجلد الحالي، أي شكل الأمر المنفذ النهائي هو find .

وسابقًا عند استخدام التابعين exec() و execFile() مررنا لهما شكل الأمر الذي نريد تنفيذه بصيغته النهائية ضمن السلسلة النصية، أما عند استدعاء spawn() فيجب تمرير المعاملات للأمر المُنفذ ضمن مصفوفة، وذلك لأن هذا التابع لا يُنشئ صَدفة جديدة قبل إنشاء وتشغيل العملية، أما إذا أردنا تمرير المعاملات مع الأمر بنفس السلسلة النصية يجب إنشاء صَدفة جديدة لتفسر ذلك.

ولنكمل معالجة تنفيذ الأمر بإضافة توابع استماع للخرج كالتالي:

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);

child.stdout.on('data', data => {
  console.log(`stdout:\n${data}`);
});

child.stderr.on('data', data => {
  console.error(`stderr: ${data}`);
});
```

كما ذكرنا سابقًا يمكن للأوامر كتابة الخرج على كل من مجرى الخرج القياسي `stdout` ومجرى خرج الأخطاء `stderr`، لذا يجب إضافة من يستمع لهما على كل مجرى باستخدام التابع `on()` ونطبع البيانات التي تُرسل ضمن ذلك الحدث إلى الطرفية.

نستمع بعدها للحدث `error` الذي سيُطلق في حال فشل تنفيذ الأمر، والحدث `close` الذي سيُطلق بعد انتهاء تنفيذ الأمر وإغلاق المجرى، ونكمل الآن كتابة البرنامج ليصبح كالتالي:

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);

child.stdout.on('data', (data) => {
  console.log(`stdout:\n${data}`);
});

child.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});

child.on('error', (error) => {
  console.error(`error: ${error.message}`);
});
```

```
child.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

لاحظ أن الاستماع لكل من الحدثين `error` و `close` يكون على كائن العملية `child` مباشرةً، ولاحظ ضمن حدث الخطأ `error` أنه يوفر لنا كائن خطأ `Error` يعبر عن المشكلة، وفي تلك الحالة سنطبع رسالة الخطأ `message` إلى الطرفية، أما ضمن حدث الإغلاق `close` تمرر نود رمز الخروج للأمر بعد تنفيذه، ومنه يمكننا معرفة نجاح أو فشل تنفيذ الأمر، فعند نجاح التنفيذ سيعيد الأمر الرمز صفر `0` وإلا سيعيد رمز خروج أكبر من الصفر.

والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج باستخدام الأمر `node`:

```
$ node findFiles.js
```

ونحصل على الخرج:

```
stdout:
.
./findFiles.js
./listFiles.js
./nodejs-logo.svg
./processNodejsImage.sh
./getNodejsImage.js

child process exited with code 0
```

يظهر لنا قائمة بكافة الملفات الموجودة ضمن المجلد الحالي، وفي آخر سطر يظهر رمز الخروج `0` ما يدل على نجاح التنفيذ، ومع أن الملفات ضمن المجلد الحالي قليلة لكن في حال نفذنا نفس الأمر ضمن مجلد آخر قد يظهر لنا قائمة طويلة جدًا من الملفات الموجودة ضمن كل المجلدات التي يمكن للمستخدم الوصول إليها، ولكن وبما أننا استخدمنا التابع `spawn()` فلا مشكلة في ذلك حيث سنعالج الخرج بأفضل طريقة ممكنة باستخدام مجاري البيانات بدلًا من تخزين الخرج كاملًا في الذاكرة ضمن مخزن مؤقت.

وبذلك نكون قد تعلمنا طرق إنشاء عملية ابن في نود لتنفيذ الأوامر الخارجية ضمن نظام التشغيل، وتتيح نود أيضًا طريقة لإنشاء عملية ابن لتنفيذ برامج نود أخرى، وذلك باستعمال التابع `fork()` وهو ما سنتعرف عليه في الفقرة التالية.

11.3 إنشاء عملية ابن باستخدام fork

تتيح نود التابع fork() المشابه للتابع `spawn()` لإنشاء عملية جديدة ابن لتنفيذ برنامج نود ما، وما يميز التابع `fork()` عن التوابع الأخرى مثل `spawn()` أو `exec()` هو إمكانية التواصل بين العملية الأب والابن، إذ إضافة لقراءة خرج الأمر الذي ننفذه باستخدام `fork()` يمكن للعملية الأب إرسال رسائل للعملية الابن والتواصل معها، ويمكن للعملية الابن أيضًا التواصل مع العملية الأب بنفس الطريقة.

وسنتعرف في هذا المثال على طريقة إنشاء عملية ابن باستخدام `fork()` والاستفادة منها في تحسين أداء التطبيق الذي نطوره، حيث وبما أن البرامج في نود تعمل ضمن عملية واحدة فالمهام التي تحتاج لمعالجة طويلة من من قبل المعالج ستعيق عمل الشيفرات التالية في باقي البرنامج، فمثلًا المهام التي تتطلب تكرار تنفيذ حلقة برمجية ما لمرات عديدة طويلة، أو تفسير ملفات كبيرة من صيغة JSON، حيث ستشكل هذه العمليات عائقًا في بعض التطبيقات وتؤثر على الأداء، فمثلًا لا يمكن لخادم ويب أن تعيق عمله مثل تلك المهام الطويلة، حيث سيمنعه ذلك من استقبال الطلبات الجديدة ومعالجتها لحين الانتهاء من تنفيذ تلك المهام، لذا سنختبر ذلك بإنشاء خادم ويب يحتوي على مسارين الأول سيُنفذ عملية تحتاج لمعالجة طويلة وتعيق عمل عملية نود للخادم، والثاني سيعيد كائن بصيغة JSON يحوي على الرسالة `hello`.

لنبدأ بإنشاء ملف جافاسكربت جديد لخادم HTTP بالاسم httpServer.js كالتالي:

```
$ nano httpServer.js
```

نبدأ بإعداد الخادم أولًا باستيراد الوحدة البرمجية `http` ثم إنشاء تابع استماع لمعالجة الطلبات الواردة، وكائن للخادم وربط تابع الاستماع معه، والآن نضيف الشيفرة التالية إلى الملف:

```
const http = require('http');

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
  console.log(`Server is running on http://${host}:${port}`);
});
```

سيكون الخادم متاحًا للوصـول على العنـوان http://localhost:8000، والآن سـنكتب دالـة مهمتهـا إعاقة عمل الخـادم عـبر حلقـة سـتُنفذ لعـدد كبـير من المـرات، ونضـيفها قبـل التـابع requestListener() كالتالي:

```
...
const port = 8000;

const slowFunction = () => {
  let counter = 0;
  while (counter < 5000000000) {
    counter++;
  }

  return counter;
}

const requestListener = function (req, res) {};
...
```

وضمن تابع معالجـة الطلب requestListener() سنسـتدعي تـابع الإعاقـة slowFunction() على المسار الفرعي، بينما سنعيد رسالة JSON على المسار الآخر كالتالي:

```
...
const requestListener = function (req, res) {
  if (req.url === '/total') {
    let slowResult = slowFunction();
    let message = `{"totalCount":${slowResult}}`;

    console.log('Returning /total results');
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    res.end(message);
  } else if (req.url === '/hello') {
    console.log('Returning /hello results');
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    res.end(`{"message":"hello"}`);
```

```
  }
};
...
```

إذا تواصلنا مع الخـادم على المسـار الفـرعي `/total` سـيُنفذ تـابع الإعاقـة `slowFunction()`، أمـا على المسار الفرعي `/hello` سنعيد الرسالة التالية بصيغة JSON بالشكل `{"message":"hello"}`، والآن نحفظ الملف ونخرج منه ثم نشغل الخادم باستخدام الأمر `node` كالتالي:

```
$ node httpServer.js
```

ليظهر لنا الرسالة التالية ضمن الخرج:

```
Server is running on http://localhost:8000
```

يمكننا بدء الاختبار الآن ولهذا نحتاج لطرفيتين إضافيتين، ففي الأولى سنستخدم الأمر `curl` لإرسـال طلب للخادم على المسار `/total` لإبطاء الخادم كالتالي:

```
$ curl http://localhost:8000/total
```

وضمن الطرفية الثانية نستخدم الأمر `curl` لإرسال طلب على المسار الآخر `/hello` كالتالي:

```
$ curl http://localhost:8000/hello
```

سيعيد الطلب الأول القيمة التالية:

```
{"totalCount":5000000000}
```

بينما سيعيد الطلب الثاني القيمة:

```
{"message":"hello"}
```

ونلاحظ أن الطلب الثاني للمسار `/hello` اكتمل بعـد انتهـاء معالجـة الطلب على المسـار `/total`، حيث أعاق تنفيذ التابع `slowFunction()` معالجة أي طلبات وتنفيذ أي شيفرات على الخادم لحين انتهائه، ويمكننا التأكد من ذلك من خرج طرفية الخادم نفسه حيث نلاحظ ترتيب إرسال الرد على تلك الطلبات:

```
Returning /total results
Returning /hello results
```

في مثل تلك الحالات يأتي دور التابع `fork()` لإنشاء عملية ابن جديدة يمكن توكيل معالجة المهام الطويلة إليها للسماح للخادم بالعمل على معالجة الطلبات الجديدة القادمة دون توقف، وسنطبق ذلـك في مثالنـا بنقـل

تابع المهمة الطويلة إلى وحدة برمجية منفصلة، حيث سيستدعيها خادم الويب لاحقًا ضمن عملية ابن منفصلة عند كل طلب إلى المسار الفرعي `/total` ويستمع إلى نتيجة التنفيذ.

نبدأ بإنشاء ملف جافاسكربت بالاسم getCount.js سيحوي على التابع `slowFunction()`:

```
$ nano getCount.js
```

ونضيف داخله ذلك التابع:

```
const slowFunction = () => {
  let counter = 0;
  while (counter < 5000000000) {
    counter++;
  }

  return counter;
}
```

وبما أننا ننوي استدعاء هذا التابع كعملية ابن باستخدام `fork()` يمكننا إضافة شيفرة للتواصل مع العملية الأب تعلمه عند انتهاء تنفيذ التابع `slowFunction()`، لهذا نضيف الشيفرة التالية التي سترسل رسالة للعملية الأب تحوي على كائن JSON لنتيجة التنفيذ ولإرسالها إلى المستخدم:

```
const slowFunction = () => {
  let counter = 0;
  while (counter < 5000000000) {
    counter++;
  }
  return counter;
}

process.on('message', (message) => {
  if (message == 'START') {
    console.log('Child process received START message');
    let slowResult = slowFunction();
    let message = `{"totalCount":${slowResult}}`;
    process.send(message);
  }
});
```

كما نلاحظ بإمكاننا الوصول للرسائل التي يُنشئها التابع `fork()` بين العملية الأب والابن عن طريق القيمة العامة للكائن `process` الذي يمثل العملية، حيث يمكننا إضافة مُستمع لحدث إرسال الرسائل `message` والتحقق ما إذا كانت الرسالة هي حدث بدء عملية المعالجة `START` الذي سيرسلها الخادم عند ورود طلب إلى المسار الفرعي `/total`، ونستجيب لتلك الرسالة بتنفيذ تابع المعالجة `slowFunction()` ثم ننشئ السلسلة النصية للرد بصيغة JSON والتي تحوي على نتيجة التنفيذ، ثم نستدعي التابع `process.send()` لإرسال رسالة للعملية الأب تعلمه بالنتيجة.

والآن نحفظ الملف ونخرج منه ونعود لملف الخادم httpServer.js للتعديل عليه وإضافة استدعاء للتابع `slowFunction()` بإنشاء عملية ابن لتنفيذ البرنامج ضمن الملف getCount.js، فنبدأ باستيراد التابع `fork()` من الوحدة البرمجية `child_process` كالتالي:

```
const http = require('http');
const { fork } = require('child_process');
...
```

ثم نزيل التابع `slowFunction()` من هذا الملف بما أننا نقلناه إلى وحدة برمجية منفصلة، ونعدل تابع معالجة الطلبات `requestListener()` ليُنشئ العملية الابن كالتالي:

```
...
const port = 8000;

const requestListener = function (req, res) {
  if (req.url === '/total') {
    const child = fork(__dirname + '/getCount');

    child.on('message', (message) => {
      console.log('Returning /total results');
      res.setHeader('Content-Type', 'application/json');
      res.writeHead(200);
      res.end(message);
    });

    child.send('START');
  } else if (req.url === '/hello') {
    console.log('Returning /hello results');
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
```

```
    res.end(`{"message":"hello"}`);
  }
};
...
```

ينتج الآن عن الطلبات الواردة إلى المسار `/total` إنشاء عملية ابن باستخدام `fork()`، حيث مررنا لهذا التابع مسار وحدة نود البرمجية التي نريد تنفيذها، وهو الملف getCount.js في حالتنا ضمن المجلد الحالي، لهذا استفدنا هذه المرة أيضًا من قيمة المتغير `__dirname` وخزنا قيمة العملية الابن ضمن المتغير `child` للتعامل معها.

أضفنا بعدها مستمعًا إلى الكائن `child` ليستقبل الرسائل الواردة من العملية الابن، وتحديدًا لاستقبال الرسالة التي سيرسلها تنفيذ الملف getCount.js الحاوية على سلسلة نصية بصيغة JSON لنتيجة تنفيذ حلقة `while`، وعند وصول تلك الرسالة نرسلها مباشرة إلى المستخدم كما هي.

ويمكننا التواصل مع العملية الابن باستدعاء التابع `send()` من الكائن `child` لإرسال رسالة لها، حيث نرسل الرسالة `START` التي سيستقبلها البرنامج ضمن العملية الابن لينفذ التابع `slowFunction()` داخله استجابة لها.

والآن نحفظ الملف ونخرج منه ونختبر الميزة التي قدمها استخدام `fork()` لخادم HTTP بتشغيل الخادم من ملف httpServer.js باستخدام الأمر `node` كالتالي:

```
$ node httpServer.js
```

وسيظهر لنا الخرج التالي:

```
Server is running on http://localhost:8000
```

وكما فعلنا سابقًا لاختبار عمل الخادم سنحتاج لطرفيتين، ففي الأولى سنستخدم الأمر `curl` لإرسال طلب للخادم على المسار `/total` والذي سيحتاج بعض الوقت للاكتمال:

```
$ curl http://localhost:8000/total
```

وضمن الطرفية الثانية نستخدم الأمر `curl` لإرسال طلب على المسار الآخر `/hello` والذي سيرسل لنا الرد هذه المرة بسرعة:

```
$ curl http://localhost:8000/hello
```

سيعيد الطلب الأول القيمة التالية:

```
{"totalCount":5000000000}
```

بينما سيعيد الطلب الثاني القيمة:

```
{"message":"hello"}
```

نلاحظ الفرق هذه المرة بأن الطلب للمسار `/hello` تم بسرعة، ويمكننا التأكد من ذلك أيضًا من الرسائل الظاهرة في طرفية الخادم:

```
Child process received START message
Returning /hello results
Returning /total results
```

حيث يظهر أن الطلب على المسار `/hello` تم استقباله بعد إنشاء العملية الابن ومعالجته قبل انتهاء عملها، وهذا بسبب نقل العملية التي تأخذ وقتا طويلًا إلى عملية ابن منفصلة واستدعائها باستخدام `fork()`، حيث بقي الخادم متفرغًا لمعالجة الطلبات الجديدة الواردة وتنفيذ شيفرات جافاسكربت، وهذا بفضل الميزة الذي يوفرها التابع `fork()` من إرسال الرسائل إلى العملية الابن للتحكم بتنفيذ العمليات ضمنها، وإمكانية قراءة البيانات المُرسلة من قبل العملية لمعالجتها ضمن العملية الأب.

11.4 خاتمة

تعرفنا في هذا الفصل على طرق مختلفة لإنشاء عملية ابن في نود، حيث تعلمنا كيف يمكن استخدام التابع `exec()` لإنشاء عملية ابن جديدة لتنفيذ أوامر الصدفة من قبل شيفرة برنامج نود، وبعدها تعرفنا على التابع `execFile()` الذي يُمكننا من تشغيل الملفات التنفيذية، ثم تعرفنا على التابع `spawn()` الذي يسمح بتنفيذ الأوامر وقراءة نتيجتها عبر مجرى للبيانات دون إنشاء صدفة لها كما يفعل التابعان `exec()` و `execFile()`، وأخيرًا تعرفنا على التابع `fork()` الذي يسمح بالتواصل بين العملية الأب والابن.

ويمكنك الرجوع إلى التوثيق الرسمي للوحدة البرمجية `child_process` من نود للتعرف عليها أكثر.

12. استخدام الوحدة fs للتعامل مع الملفات

كثيرًا ما نحتاج للتعامل مع نظام الملفات، فمثلًا لتخزين بعض الملفات بعد تنزيلها، أو لترتيب بعض البيانات ضمن مجلدات أو لقراءة الملفات للتعامل مع محتوياتها ضمن بعض التطبيقات، وحتى تطبيقات النظم الخلفية أو أدوات واجهة سطر الأوامر CLI قد تحتاج أحيانًا لحفظ بعض البيانات إلى ملفات، والتطبيقات التي تتعامل مع البيانات تحتاج أحيانًا لتصديرها بمختلف الصيغ مثل JSON أو CSV أو ملفات برنامج إكسل، فكل تلك المتطلبات تحتاج للتعامل مع نظام الملفات ضمن نظام التشغيل التي تعمل عليه.

توفر نود طريقة برمجية للتعامل مع الملفات باستخدام الوحدة البرمجية fs، وهي اختصار لجملة "نظام الملفات" أو "file system" حيث تحتوي على العديد من التوابع التي نحتاجها لقراءة الملفات أو الكتابة إليها أو حذفها، وهذه المزايا تجعل من لغة جافاسكربت لغة مفيدة لاستخدامها ضمن تطبيقات النظم الخلفية و أدوات سطر الأوامر.

سنتعرف في هذا الفصل على الوحدة البرمجية `fs` وسنستخدمها لقراءة الملفات وإنشاء ملفات جديدة والكتابة إليها وحذف الملفات وحتى نقل الملفات من مجلد إلى آخر، حيث توفر الوحدة البرمجية `fs` توابع للتعامل مع الملفات بالطريقتين المتزامنة synchronously واللامتزامنة asynchronously وباستخدام مجاري البيانات streams، حيث سنستخدم في هذا الفصل الطريقة اللامتزامنة باستخدام الوعود Promises وهي الطريقة الأكثر استخدامًا.

12.1 قراءة الملفات باستخدام readFile()

سنطور في هذه الفقرة برنامجًا في نود لقراءة الملفات، وسنستعين بالتابع readFile() الذي توفره الوحدة البرمجية `fs` في نود لقراءة محتوى ملف معين وتخزينه ضمن متغير ثم طباعته إلى الطرفية.

سنبدأ بإعداد المجلد الذي سيحوي على ملفات الأمثلة المستخدمة في هذا الفصل ونُنشئ لذلك مجلدًا جديدًا بالاسم `node-files` كالتالي:

```
$ mkdir node-files
```

وندخل لذلك المجلد باستخدام الأمر `cd`:

```
$ cd node-files
```

ننشئ داخل المجلد ملفين الأول هو الملف الذي سنحاول قراءته باستخدام البرنامج، والثاني هو ملف جافاسكربت للبرنامج الذي سنطوره، ونبدأ بإنشاء ملف جديد يحوي المحتوى النصي `greetings.txt`، وهنا سنُنشئ الملف عن طريق سطر الأوامر كالتالي:

```
$ echo "hello, hola, bonjour, hallo" > greetings.txt
```

في الأمر السابق سيطبع الأمر `echo` النص المُمرر له إلى الطرفية، واستخدمنا المعامل `>` لإعادة توجيه خرج الأمر `echo` إلى الملف النصي الجديد `greetings.txt`.

والآن ننشئ ملف جافاسكربت جديد للبرنامج بالاسم `readFile.js` ونفتحه باستخدام أي محرر نصوص، حيث سنستخدم ضمن أمثلتنا محرر النصوص `nano` كالتالي:

```
$ nano readFile.js
```

يتكون البرنامج الذي سنكتبه من ثلاث أقسام رئيسية، حيث نبدأ أولًا باستيراد الوحدة البرمجية التي تحوي توابع التعامل مع الملفات كالتالي:

```
const fs = require('fs').promises;
```

تحوي الوحدة `fs` كافة التوابع المستخدمة في التعامل مع نظام الملفات، ونلاحظ كيف استوردنا منها الجزء `.promises` حيث كانت طريقة كتابة الشيفرة اللامتزامنة سابقًا ضمن الوحدة `fs` عبر استخدام دوال رد النداء callbacks، ولاحقًا وبعد أن انتشر استخدام الوعود كطريقة بديلة أضاف فريق التطوير في نود دعمًا لها ضمن الوحدة `fs`، حيث وبدءًا من الإصدار رقم 10 من نود أضيفت الخاصية `promises` ضمن كائن الوحدة البرمجية `fs` والتي تحوي التوابع التي تدعم طريقة الوعود، بينما بقي عمل الوحدة البرمجية `fs` الأساسية كما هي سابقًا باستخدام توابع رد النداء لدعم البرامج التي تستخدم الطريقة القديمة، وفي أمثلتنا سنستخدم نسخة التوابع التي تعتمد على الوعود.

في القسم الثاني من البرنامج سنضيف دالة لامتزامنة لقراءة محتوى الملف، حيث يمكن تعريف الدوال اللامتزامنة في جافاسكربت بإضافة الكلمة `async` في بدايتها، وبذلك نستطيع ضمن التابع انتظار نتيجة كائنات الوعود باستخدام الكلمة `await` مباشرةً بدلًا من ربط العمليات المتتالية باستخدام التابع `.then()`.

والآن نعرف الدالة `readFile()` التي تقبل سلسلة نصية `filePath` تمثِّل مسار الملف الذي نود قراءته، حيث سنستعين بتوابع الوحدة `fs` لقراءة محتوى الملف المطلوب وتخزينه ضمن متغير باستخدام صيغة `async/await` كالتالي:

```
const fs = require('fs').promises;

async function readFile(filePath) {
  try {
    const data = await fs.readFile(filePath);
    console.log(data.toString());
  } catch (error) {
    console.error(`Got an error trying to read the file: $
{error.message}`);
  }
}
```

يمكن التقاط الأخطاء التي قد يرميها استدعاء التابع `fs.readFile()` باستخدام try...catch، حيث نستدعي التابع `fs.readFile()` ضمن جسم `try` ثم نخزن النتيجة ضمن المتغير `data`، ويقبل ذلك التابع معاملًا وحيدًا إجباريًا وهو مسار الملف الذي نود قراءته، ويعيد كائن مخزن مؤقت `buffer` كنتيجة لعملية القراءة حيث يمكن لهذا الكائن أن يحوي أي نوع من الملفات، ولكي نطبع ذلك المحتوى إلى الطرفية يجب تحويله إلى سلسلة نصية باستخدام التابع `toString()` من كائن المخزن المؤقت.

وفي حال رمي خطأ ما فالسبب يكون إما لعدم وجود الملف الذي نريد قراءته، أو لأن المستخدم لا يملك إذنًا لقراءته، ففي هذه الحالة سنطبع رسالة خطأ إلى الطرفية.

أما القسم الثالث والأخير من البرنامج هو استدعاء دالة قراءة الملف مع تمرير اسم الملف `greetings.txt` ليصبح البرنامج كالتالي:

```
const fs = require('fs').promises;

async function readFile(filePath) {
  try {
    const data = await fs.readFile(filePath);
    console.log(data.toString());
  } catch (error) {
    console.error(`Got an error trying to read the file: $
{error.message}`);
  }
```

```
}

readFile('greetings.txt');
```

نحفظ الملف ونخرج منه وفي حال كنت تستخدم أيضًا محرر النصوص nano يمكنك الخروج بالضغط على الاختصار CTRL+X، وعند تنفيذ البرنامج سيقرأ المحتوى النصي للملف `greetings.txt` ويطبع محتواه إلى الطرفية، والآن ننفذ البرنامج عبر الأمر `node` لنرى النتيجة:

```
$ node readFile.js
```

بعد تنفيذ الأمر سيظهر الخرج التالي:

```
hello, hola, bonjour, hallo
```

وبذلك نكون قد استخدمنا التابع `readFile()` من الوحدة `fs` لقراءة محتوى الملف باستخدام صيغة `async/await`.

> انتبه، إذا كنت تستخدم إصدارًا قديمًا من نود وحاولت استخدام الوحدة `fs` بالطريقة السابقة سيظهر لك رسالة التحذير التالية:
>
> (node:13085) ExperimentalWarning: The fs.promises API is experimental
>
> حيث أن الخاصية `promises` من الوحدة `fs` تم اعتمادها رسميًا منذ الإصدار 10 من نود، وقبل ذلك كانت في المرحلة التجريبية وهذا سبب رسالة التحذير السابقة، ولاحقًا وتحديدًا ضمن إصدار نود رقم 12.6 أصبحت التوابع ضمن تلك الخاصية مستقرة وأزيلت رسالة التحذير تلك.

الآن وبعد أن تعرفنا على طريقة قراءة الملفات باستخدام الوحدة `fs` سنتعلم في الفقرة التالية طريقة إنشاء الملفات الجديدة وكتابة المحتوى النصي إليها.

12.2 كتابة الملفات باستخدام writeFile()

سنتعلم في هذه الفقرة طريقة كتابة الملفات باستخدام التابع writeFile() من الوحدة البرمجية `fs`، وذلك بكتابة ملف بصيغة CSV يحوي على بيانات لفاتورة شراء، حيث سنبدأ بإنشاء ملف جديد وإضافة ترويسات عناوين الأعمدة له، ثم سنتعلم طريقة إضافة بيانات جديدة إلى نهاية الملف.

نبدأ أولًا بإنشاء ملف جافاسكربت جديد للبرنامج ونفتحه باستخدام محرر النصوص كالتالي:

```
$ nano writeFile.js
```

ونستورد الوحدة `fs` كالتالي:

```
const fs = require('fs').promises;
```

وسنستخدم في هذا المثال أيضًا صيغة `async/await` لتعريف دالـتين، الأولى لإنشـاء ملـف CSV جديـد والثانية لكتابة بيانات جديدة إليه.

نفتح الملف ضمن محرر النصوص ونضيف الدالة التالي:

```
const fs = require('fs').promises;

async function openFile() {
  try {
    const csvHeaders = 'name,quantity,price'
    await fs.writeFile('groceries.csv', csvHeaders);
  } catch (error) {
    console.error(`Got an error trying to write to a file: $
{error.message}`);
  }
}
```

نُعرّف المتغير `csvHeaders` والذي يحتـوي على عنـاوين رؤوس الأعمـدة لملـف CSV، ثم نسـتدعي التـابع `writeFile()` من وحدة `fs` لإنشاء ملف جديد وكتابة البيانـات إليـه، حيث أن المعامـل الأول المُمـرر لـه هـو مسار الملف الجديد، وإذا مررنا اسم الملف فقط فسيُنشَأ الملف الجديد ضمن المسار الحـالي لتنفيـذ البرنـامج، وأما المعامل الثـاني المُمـرر هـو البيانـات الـتي نريـد كتابتهـا ضـمن الملـف، وفي حالتنـا هي عنـاوين الأعمـدة الموجودة ضمن المتغير `csvHeaders`.

والآن نضيف الدالة الثانية ومهمتها إضافة بيانات جديدة ضمن ملف الفاتورة كالتالي:

```
const fs = require('fs').promises;

async function openFile() {
  try {
    const csvHeaders = 'name,quantity,price'
    await fs.writeFile('groceries.csv', csvHeaders);
  } catch (error) {
    console.error(`Got an error trying to write to a file: $
{error.message}`);
  }
}
```

```
async function addGroceryItem(name, quantity, price) {
  try {
    const csvLine = `\n${name},${quantity},${price}`
    await fs.writeFile('groceries.csv', csvLine, { flag: 'a' });
  } catch (error) {
    console.error(`Got an error trying to write to a file: $
{error.message}`);
  }
}
```

عرفنا الدالة اللامتزامنة `addGroceryItem()` التي تقبل ثلاثة مُعاملات، وهي اسم المنتج والكمية والسعر للقطعة الواحدة منه، ويتم إنشاء السطر الجديد الذي نود كتابته إلى الملف باستخدام قالب نص template literal وتخزينه ضمن المتغير `csvLine`، ثم نستدعي التابع `writeFile()` كما فعلنا سابقًا ضمن التابع الأول `openFile()`، ولكن هذه المرة سنمرر كائن جافاسكربت كمعامل ثالث يحتوي على المفتاح `flag` بالقيمة `a`، وتعبر تلك القيمة عن الرايات المستخدمة للتعامل مع نظام الملفات، والراية `a` هنا تخبر نود بأننا نريد إضافة ذلك المحتوى إلى الملف، وليس إعادة كتابة محتوى الملف كاملًا، وفي حال لم نمرر أي راية عند كتابة الملف كما فعلنا ضمن الدالة الأولى فإن القيمة الافتراضية هي الراية `w` والتي تعني إنشاء ملف جديد في حال لم يكن الملف موجودًا، وإذا كان موجودًا سيتم تبديله وإعادة كتابة محتواه كاملًا، ويمكنك الرجوع إلى التوثيق الرسمي لتلك الرايات من نود للتعرف عليها أكثر.

والآن لننهي كتابة البرنامج باستدعاء الدوال التي عرّفناها كالتالي:

```
...
async function addGroceryItem(name, quantity, price) {
  try {
    const csvLine = `\n${name},${quantity},${price}`
    await fs.writeFile('groceries.csv', csvLine, { flag: 'a' });
  } catch (error) {
    console.error(`Got an error trying to write to a file: $
{error.message}`);
  }
}

(async function () {
  await openFile();
  await addGroceryItem('eggs', 12, 1.50);
```

```
  await addGroceryItem('nutella', 1, 4);
})();
```

وبما أن الدوال التي سنستدعيها لامتزامنة، فيمكننا تغليفها بدالة لامتزامنة واستدعاءها مباشرة كي نستطيع استخدام `await` لانتظار إكمال تنفيذها، وذلك لأنه لا يمكن ضمن إصدار نود الذي نستخدمه حاليًا استخدام `await` مباشرة ضمن النطاق العام global scope، بل حصرًا ضمن دوال لامتزامنة تُستخدم ضمن تعريفها الكلمة `async`، ولا حاجة لتسمية تلك الدالة ويمكننا تعريفها كدالة مجهولة لأن الغرض منها فقط التغليف والتنفيذ المباشر ولن نشير إليها من أي مكان آخر.

وبما أن كلا الدالتين `openFile()` و `addGroceryItem()` لا متزامنين فبدون انتظار نتيجة استدعاء الدالة الأولى ثم استدعاء الثانية لا يمكن ضمان ترتيب التنفيذ وبالتالي ترتيب المحتوى ضمن الملف الذي نريد إنشاءه، لذلك عرفنا دالة التغليف تلك الغير متزامنة بين قوسين وأضفنا قوسي الاستدعاء في النهاية قبل الفاصلة المنقوطة كي لاستدعائها مباشرةً، وتُدعى تلك الصيغة بصيغة التنفيذ المباشر لدالة Immediately-Invoked Function Expression أو IIFE، وباستخدام تلك الصيغة في مثالنا نضمن احتواء ملف CSV الجديد على الترويسات بدايةً ثم أول سطر للمنتج `eggs` وبعده المنتج الثاني `nutella`.

والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج باستخدام الأمر `node`:

```
$ node writeFile.js
```

لن نلاحظ أي خرج من التنفيذ ولكن سنلاحظ إنشاء ملف جديد ضمن المجلد الحالي ويمكن معاينة محتوى الملف `groceries.csv` باستخدام الأمر `cat` كالتالي:

```
$ cat groceries.csv
```

ليظهر الخرج التالي:

```
name,quantity,price
eggs,12,1.5
nutella,1,4
```

أنشأت الدالة `openFile()` ملف CSV وأضافت الترويسات له، ثم أضافت استدعاءات الدالة `addGroceryItem()` التي تليها سطرين من البيانات إلى ذلك الملف، وبذلك نكون قد تعلمنا طريقة استخدام التابع `writeFile()` لإنشاء الملفات الجديدة والتعديل على محتواها.

سنتعلم في الفقرة التالية كيف يمكننا حذف الملفات في حال أردنا إنشاء ملفات مؤقتة مثلًا، أو لإزالة بعض الملفات لتوفير مساحة التخزين على الجهاز.

12.3 حذف الملفات باستخدام unlink()

سنتعلم في هذه الفقرة طريقة حذف الملفات باستخدام التابع unlink() من الوحدة البرمجية fs، حيث سنكتب برنامجًا لحذف الملف groceries.csv الذي أنشأناه في الفقرة السابقة.

نبدأ بإنشاء ملف جافاسكربت جديد بالاسم deleteFile.js نُعرف ضمنه الدالة اللامتزامنة deleteFile() التي تقبل مسار الملف المراد حذفه، وبدورها ستمرر ذلك المعامل إلى التابع unlink() والذي سيحذف ذلك الملف من نظام الملفات كالتالي:

```
const fs = require('fs').promises;

async function deleteFile(filePath) {
  try {
    await fs.unlink(filePath);
    console.log(`Deleted ${filePath}`);
  } catch (error) {
    console.error(`Got an error trying to delete the file: $
{error.message}`);
  }
}

deleteFile('groceries.csv');
```

لن تُنقل الملفات المحذوفة باستخدام التابع unlink() إلى سلة المحذوفات بل ستُحذف نهائيًا من نظام الملفات، لذا تلك العملية لا يمكن الرجوع عنها ويجب الحذر والتأكد من الملفات التي نحاول حذفها قبل تنفيذ البرنامج.

والآن نخرج من الملف وننفذه كالتالي:

```
$ node deleteFile.js
```

ليظهر الخرج التالي:

```
Deleted groceries.csv
```

نستعرض الملفات الموجودة حاليًا بعد التنفيذ للتأكد من نجاح عملية الحذف باستخدام الأمر ls كالتالي:

```
$ ls
```

ليظهر لنا الملفات التالية:

```
deleteFile.js   greetings.txt   readFile.js     writeFile.js
```

نلاحظ حذف الملف بنجاح باستخدام التابع unlink()، وبذلك نكون قد تعلمنا طريقة قراءة وكتابة وحذف الملفات، وسنتعلم في الفقرة التالية كيف يمكن نقل الملفات من مجلد لآخر، لنكون بذلك قد تعلمنا كافة العمليات التي تسمح بإدارة الملفات عن طريق نود.

12.4 نقل الملفات باستخدام rename()

تُستخدم المجلدات لتنظيم وترتيب الملفات معًا، لذا من المفيد تعلم طريقة نقل تلك الملفات برمجيًا، حيث يتم ذلك في نود باستخدام التابع rename() من الوحدة fs، وسنتعلم طريقة استخدامه بنقل الملف السابق greetings.txt إلى مجلد جديد مع إعادة تسميته.

نبدأ بإنشاء ذلك مجلد جديد بالاسم test-data ضمن المجلد الحالي كالتالي:

```
$ mkdir test-data
```

ونُنشئ نسخة عن الملف greetings.txt بتنفيذ أمر النسخ cp كالتالي:

```
$ cp greetings.txt greetings-2.txt
```

ثم نُنشئ ملف جافاسكربت للبرنامج كالتالي:

```
$ nano moveFile.js
```

ونُعرف ضمنه الدالة moveFile() لنقل الملف، والتي ستستدعي بدورها التابع rename() الذي يقبل مسار الملف المراد نقله كمعامل أول، ثم المسار الجديد الوجهة كمعامل ثانٍ، ففي حالتنا نريد استخدام الدالة moveFile() لنقل الملف الجديد greetings-2.txt إلى المجلد الذي أنشأناه test-data مع إعادة تسمية ذلك الملف إلى salutations.txt، ولذلك نضيف الشيفرة التالية:

```
const fs = require('fs').promises;

async function moveFile(source, destination) {
  try {
    await fs.rename(source, destination);
    console.log(`Moved file from ${source} to ${destination}`);
  } catch (error) {
    console.error(`Got an error trying to move the file: $
{error.message}`);
  }
```

```
}

moveFile('greetings-2.txt', 'test-data/salutations.txt');
```

كما ذكرنا سابقًا فالتابع `rename()` يقبل معاملين هما المسار المصدر والوجهة لنقل الملف، ويمكن استخدام هذا التابع إما لنقل الملفات من مجلد لآخر أو لإعادة تسمية الملفات، أو نقل وإعادة تسمية ملف ما معًا، وهو ما نريد تنفيذه في مثالنا.

والآن نحفظ الملف ونخرج منه وننفذه باستخدام الأمر `node` كالتالي:

```
$ node moveFile.js
```

ليظهر الخرج التالي:

```
Moved file from greetings-2.txt to test-data/salutations.txt
```

نستعرض الملفات الموجودة حاليًا بعد التنفيذ للتأكد من نجاح عملية النقل باستخدام الأمر `ls` كالتالي:

```
$ ls
```

ليظهر لنا الملفات والمجلدات التالية:

```
deleteFile.js   greetings.txt   moveFile.js     readFile.js     test-
data        writeFile.js
```

ونستخدم الأمر `ls` مجددًا لعرض الملفات ضمن المجلد الوجهة `test-data`:

```
$ ls test-data
```

ليظهر لنا الملف الذي نقلناه:

```
salutations.txt
```

وبذلك نكون قد تعلمنا كيف يمكن استخدام التابع `rename()` لنقل الملفات من مجلد لآخر مع إعادة تسمية الملف ضمن نفس العملية.

12.5 خاتمة

تعرفنا في هذا الفصل على مختلف عمليات إدارة الملفات ضمن نود، بداية بقراءة محتوى الملفات باستخدام `readFile()` ثم إنشاء ملفات جديدة وكتابة البيانات إليها باستخدام `writeFile()` ثم طريقة حذف الملفات باستخدام `unlink()` ونقلها وإعادة تسميتها باستخدام `rename()`.

إنَّ التعامل مع الملفات من المهام الضرورية في نود فقد تحتاج الـبرامج أحيانًـا إلى تصـدير بعض الملفـات للمستخدم أو تخزين البيانات الخاصة بها ضمن الملفات لاستعادتها لاحقًا، ولذلك توفر الوحدة البرمجيــة `fs` في نود كل التوابع الضرورية للتعامل مع الملفات في نود، ويمكنك الرجوع إلى التوثيق الرسمي للوحدة البرمجية `fs` من نود للتعرف عليها أكثر.

13. التعامل مع طلبات HTTP

تحتاج معظم التطبيقات حاليًا إلى التواصل مع بعض الخوادم لجلب البيانات منها أو لإتمام بعض المهام، فمثلًا في تطبيق ويب لشراء الكتب سيحتاج للتواصل مع خادم إدارة طلبات الزبائن وخادم مستودع الكتب وخادم إتمام الدفع، حيث تتواصل تلك الخدمات مع بعضها عن طريق الويب عبر الواجهات البرمجية API وتتبادل البيانات برمجيًا.

توفر نود دعمًا للتواصل عن طريق طلبات HTTP مع واجهات API عبر الويب، فتتيح الوحدة البرمجية `http` والوحدة `https`، حيث تحتوي كل منهما على التوابع اللازمة لإنشاء خادم HTTP لمعالجة الطلبات الواردة إلى الخادم، وتوابع لإنشاء طلبات HTTP وإرسالها إلى الخوادم الأخرى، حيث تسمح هاتين الميزتين بتطوير تطبيقات ويب حديثة تعتمد على الواجهات البرمجية API للتواصل بينها، ولا حاجة لتثبيت أي وحدة برمجية خارجية حيث تأتي تلك الوحدات جاهزة مع نود افتراضيًا.

سنتعلم في هذا الفصل كيف يمكننا الاستفادة من الوحدة https لإرسال طلبات HTTP بمثال عن التعامل مع خادم JSON Placeholder وهو واجهة برمجية API وهمية تستخدم في عمليات التدريب والاختبار، حيث سنتعلم طريقة إرسال طلب HTTP لطلب البيانات من نوع `GET`، ثم سنتعرف على طرق تخصيص الطلب المرسل كإضافة الترويسات، وسنتعرف أيضًا على الطلبات بمختلف أنواعها مثل `POST` و `PUT` و `DELETE` والتي تستخدم لتعديل البيانات على الخوادم الأخرى.

13.1 إرسال طلب من نوع GET

إن أردنا طلب بيانات من خادم ويب ما عبر واجهته البرمجية API نرسل إليه عادة طلب HTTP من نوع `GET`، ففي هذه الفقرة سنتعلم طريقة إرسال تلك الطلبات في نود وتحديدًا لجلب مصفوفة بيانات بصيغة JSON تحتوي على بيانات حسابات شخصية لمستخدمين وهميين من واجهة برمجية API متاحة للعموم، حيث

تحوي الوحدة البرمجية `https` على تابعين يمكن استخدامهما لإرسال طلبات من نوع `GET` هما التابع get()، والتابع request() الذي يمكن استخدامه لإرسال طلبات من أنواع متعددة أخرى كما سنتعلم لاحقًا، ولنبدأ أولًا بالتعرف على التابع `get()`.

13.1.1 إرسال الطلبات باستخدام التابع get()

صيغة استخدام التابع `get()` تكون كالتالي:

```
https.get(URL_String, Callback_Function) {
    Action
}
```

حيث نمرر له سلسلة نصية كمعامل أول تحوي المسار الذي سنرسل الطلب إليه، والمعامل الثاني يكون دالة رد النداء callback function لمعالجة نتيجة الطلب.

سنبدأ بإنشاء مجلد جديد للمشروع سيحوي الأمثلة التي سنكتبها ضمن هذا الفصل كالتالي:

```
$ mkdir requests
```

وندخل إلى المجلد:

```
$ cd requests
```

ننشئ ملف جافاسكربت جديد ونفتحه باستخدام أي محرر نصوص حيث سنستخدم في أمثلتنا محرر نانو nano كالتالي:

```
$ nano getRequestWithGet.js
```

ونبدأ باستيراد الوحدة `https` كالتالي:

```
const https = require('https');
```

> يوجد في نود وحدتين برمجيتين هما `http` و `https` تحويان على نفس التوابع التي تعمل بنفس الطريقة، والفرق بينها أن التوابع ضمن `https` ترسل الطلبات عبر طبقة أمان النقل Transport Layer Security أو TLS/SSL، وسنرسل الطلبات في أمثلتنا عبر HTTPS لذا سنستخدم تلك التوابع من الوحدة `https`، بينما لو كنا سنرسل الطلبات عبر HTTP فيجب استخدام توابع الوحدة `http` بدلًا منها.

نبدأ بكتابة شيفرة إرسال طلب `GET` إلى الواجهة البرمجية API لجلب بيانات المستخدمين، حيث سنرسل طلبًا إلى واجهة JSON Placeholder وهي واجهة برمجية متاحة للاستخدام العام لأغراض الاختبار، ولا تتأثر

البيانات على ذلك الخادم بالطلبات المرسلة فمهمته فقط محاكاة عمل خادم حقيقي، حيث سيرجع لنا دومًا بيانات وهمية طالما أن الطلب المرسل إليه سليم، لنبدأ بكتابة الشيفرة التالية:

```
const https = require('https');


let request = https.get('https://jsonplaceholder.typicode.com/users?
_limit=2', (res) => { });
```

كما ذكرنا سابقًا يقبل التابع `get()` معاملين وهما المسار الوجهة للطلب URL كسلسلة نصية للواجهة البرمجية، ودالة رد نداء لمعالجة نتيجة طلب HTTP الواردة، حيث يمكن استخراج البيانات من الرد ضمن دالة رد النداء.

يحمل طلب HTTP على رمز الحالة status code وهو عدد يشير إلى نجاح الطلب من عدمه، فمثلًا إذا كانت قيمة الرمز بين 200 و 299 فالطلب ناجح، أما إذا كان بين 400 و 599 فهناك خطأ ما، وفي مثالنا الرد من الواجهة البرمجية يجب أن يحتوي على رمز الحالة 200 إن نجح وهو أول ما سنتحقق منه ضمن تابع رد النداء لمعالجة الطلب كالتالي:

```
const https = require('https');


let request = https.get('https://jsonplaceholder.typicode.com/users?
_limit=2', (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: $
{res.statusCode}`);
    res.resume();
    return;
  }
});
```

يحوي كائن الرد `res` المُمرر لدالة رد النداء على الخاصية `statusCode` والتي تمثِّل قيمة رمز الحالة، وإذا لم تكن قيمته تساوي 200 سنطبع رسالة خطأ إلى الطرفية ونخرج مباشرةً.

نلاحظ استدعاء التابع `res.resume()` من كائن الرد وهي طريقة لتحسين أداء البرنامج فعند إرسال طلبات HTTP في نود يتم عادة معالجة البيانات المرسلة ضمن الطلب كاملةً، أما عند استدعائنا للتابع `res.resume()` فإننا نخبر نود بتجاهل البيانات ضمن مجرى كائن الرد، وهي طريقة أسرع من لو تُركت تلك البيانات لالتقاطها في مرحلة كنس المهملات garbage collection التي تتم دوريًا لتفريغ الذاكرة المستخدمة من قبل التطبيق.

والآن بعد أن تحققنا من رمز الحالة للرد سنبدأ بقراءة البيانات الواردة حيث تتوفر البيانات ضمن كائن مجرى الرد على دفعات، ويمكننا قراءتها بالاستماع إلى الحدث data من كائن الرد ثم تجميعها معًا ثم تحليلها بصيغة JSON لنتمكن من استخدامها ضمن التطبيق، لذلك سنضيف الشيفرة التالية ضمن تابع رد النداء:

```
const https = require('https');

let request = https.get('https://jsonplaceholder.typicode.com/users?
_limit=2', (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: $
{res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Retrieved all data');
    console.log(JSON.parse(data));
  });
});
```

عرفنا متغيرًا جديدًا بالاسم data والذي يحتوي على سلسلة نصية فارغة، حيث يمكننا تجميع البيانات الواردة إما على شكل مصفوفة من الأعداد تمثل البيانات للبايتات المكونة لها، أو على شكل سلسلة نصية وهو ما سنستخدمه في مثالنا لسهولة تحويل السلسلة النصية الناتجة عن عملية التجميع إلى كائن جافاسكربت.

نضيف بعد ذلك تابع الاستماع للبيانات الواردة على دفعات من الحدث data ونجمع البيانات كلها ضمن المتغير السابق data، ويمكننا التأكد من انتهاء دفعات البيانات الواردة عند إطلاق حدث الإغلاق close من كائن الرد، وبعدها يمكننا تحويل السلسلة النصية بصيغة JSON ضمن المتغير data وطباعة القيمة النهائية إلى الطرفية، وبذلك نكون قد أكملنا كتابة عملية إرسال طلب إلى واجهة برمجية ستُرسل بدورها مصفوفة من بيانات حسابات شخصية لثلاثة مستخدمين بصيغة JSON ونقرأ الرد الوارد.

بقي لدينا إضافة معالجة لحالة رمي خطأ في حال لم نتمكن من إرسال الطلب لسبب ما، كحالة عـدم وجـود اتصال بالإنترنت كالتالي:

```
...
  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Retrieved all data');
    console.log(JSON.parse(data));
  });

});

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a request: $
{err.message}`);
});
```

عند حدوث خطأ في عملية الإرسال سنتلقى الحدث `error` من كائن الطلب، وإذا لم نسـتمع لهـذا الحـدث فسيرمى الخطأ الناتج ما يؤدي لإيقاف عمـل البرنـامج، لـذلك نضـيف دالـة اسـتماع للحـدث `error` على كـائن الطلب باستخدام التابع `on()` والـذي سـيطبع رسـالة الخطـأ الـوارد إلى الطرفيـة، وبـذلك نكـون قـد انتهينـا من كتابة البرنامج.

والآن نحفظ الملف ونخرج منه وننفذه باستخدام الأمر `node` كالتالي:

```
$ node getRequestWithGet.js
```

نحصل على الخرج التالي الذي يمثل الرد الوارد على الطلب المُرسل:

```
Retrieved all data
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
```

```
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
    phone: '1-770-736-8031 x56442',
    website: 'hildegard.org',
    company: {
      name: 'Romaguera-Crona',
      catchPhrase: 'Multi-layered client-server neural-net',
      bs: 'harness real-time e-markets'
    }
  },
  {
    id: 2,
    name: 'Ervin Howell',
    username: 'Antonette',
    email: 'Shanna@melissa.tv',
    address: {
      street: 'Victor Plains',
      suite: 'Suite 879',
      city: 'Wisokyburgh',
      zipcode: '90566-7771',
      geo: [Object]
    },
    phone: '010-692-6593 x09125',
    website: 'anastasia.net',
    company: {
      name: 'Deckow-Crist',
      catchPhrase: 'Proactive didactic contingency',
      bs: 'synergize scalable supply-chains'
    }
  }
]
```

بـذلك نكـون قـد أرسـلنا طلبًـا من نـوع GET بنجـاح باسـتخدام مكتبـات نـود فقـط، حيث أن التـابع الـذي استخدمناه get() يوجد في نود بسبب كثرة الحاجة لإرسال الطلبـات من نـوع GET، بينمـا الطريقـة الأساسـية لإرسال الطلبات هي باستخدام التابع request() والذي يمكنه إرسال أي نوع من الطلبات، وهـو مـا سـنتعرف عليه في القسم التالي حيث سنستخدمه لإرسال طلب من نوع GET.

13.2 إرسال الطلبات باستخدام التابع request()

يمكن استخدام التابع request() بعدة صيغ والصيغة التي سنستخدمها في أمثلتنا هي كالتالي:

```
https.request(URL_String, Options_Object, Callback_Function) {
    Action
}
```

حيث نمرر له سلسلة نصية كمعامل أول تحتـوي على مسـار الواجهـة البرمجيـة API الـذي سنرسـل الطلب إليه، والمعامل الثاني هو كائن جافاسكربت يحتوي على عدة خيارات للطلب المرسـل، والمعامـل الأخـير المُمـرر هو دالة رد النداء callback لمعالجة نتيجة الطلب.

نبدأ بإنشاء ملف جافاسكربت جديد بالاسم getRequestWithRequest.js:

```
$ nano getRequestWithRequest.js
```

سنكتب برنامجًا مشابهًا لما كتبناه في القسم السـابق ضـمن الملـف getRequestWithGet.js، حيث نبدأ باستيراد الوحدة https كالتالي:

```
const https = require('https');
```

ثم نعرف كائن جافاسكربت يحتوي على الخاصية method كالتالي:

```
const https = require('https');

const options = {
  method: 'GET'
};
```

تعبر الخاصية method ضمن كائن خيارات التـابع request() عن نـوع الطلب الـذي نريـد إرسـاله، والآن نُرسل الطلب كما فعلنا سابقًا لكن مع بعض الاختلافات كالتالي:

```
...
let request =
https.request('https://jsonplaceholder.typicode.com/users?_limit=2',
options, (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: $
{res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Retrieved all data');
    console.log(JSON.parse(data));
  });
});

request.end();

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a request: $
{err.message}`);
});
```

مررنا مسار الوجهة للطلب كمعامل أول للتابع `request()` ثم كائن خيارات HTTP كمعامل ثـاني، وبعـدها دالة رد النداء لمعالجة الرد، حيث حددنا نوع الطلب المرسل كطلب `GET` ضمن كائن الخيـارات `options` الـذي عرفناه سابقًا، وبقي دالة رد النداء لمعالجة الطلب كما هو في المثال السابق، وأضفنا استدعاءً للتابع `end()` من كائن الطلب `request`، حيث يجب استدعاء هذا التـابع عنـد إرسـال الطلبـات باسـتخدام `request()` لإتمـام الطلب وإرساله، وفي حال لم نستدعيه فلن يُرسل الطلب ويبقى نود ينتظر منا إضافة بيانات جديدة إلى الطلب.

والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج:

```
$ node getRequestWithRequest.js
```

ليظهر الخرج التالي كما المثال السابق تمامًا:

```
Retrieved all data
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
    phone: '1-770-736-8031 x56442',
    website: 'hildegard.org',
    company: {
      name: 'Romaguera-Crona',
      catchPhrase: 'Multi-layered client-server neural-net',
      bs: 'harness real-time e-markets'
    }
  },
  {
    id: 2,
    name: 'Ervin Howell',
    username: 'Antonette',
    email: 'Shanna@melissa.tv',
    address: {
      street: 'Victor Plains',
      suite: 'Suite 879',
      city: 'Wisokyburgh',
      zipcode: '90566-7771',
      geo: [Object]
```

```
    },
    phone: '010-692-6593 x09125',
    website: 'anastasia.net',
    company: {
      name: 'Deckow-Crist',
      catchPhrase: 'Proactive didactic contingency',
      bs: 'synergize scalable supply-chains'
    }
  }
]
```

وبذلك نكون قد تعرفنا على طريقة استخدام التابع `request()` لإرسال الطلبات من نوع `GET` وهو تابع أقوى من التابع السابق `get()` حيث يسمح بتخصيصات عدة على الطلب المرسل كتحديد نوعه وخيارات أخرى سنتعرف عليها في الفقرة التالية.

13.3 تخصيص خيارات HTTP للتابع request()

يمكن استخدام التابع `request()` لإرسال طلبات HTTP دون تمرير عنوان مسار الوجهة للطلب كمعامل أول بل بتمريره ضمن كائن الخيارات `options` لتصبح صيغة استدعاء التابع كالتالي:

```
https.request(Options_Object, Callback_Function) {
    Action
}
```

في هذه الفقرة سنستخدم هذه الصيغة للتركيز على إعداد وتخصيص خيارات التابع `request()`، لذا نعود إلى ملف المثال السابق `getRequestWithRequest.js` ونعدله بأن نزيل المسار URL المُمرر للتابع `request()` لتصبح المعاملات المُمررة له هي كائن الخيارات `options` ودالة رد النداء فقط كالتالي:

```
const https = require('https');

const options = {
  method: 'GET',
};

let request = https.request(options, (res) => {
...
```

لنضيف الخيارات الجديدة إلى الكائن `options` كالتالي:

```
const https = require('https');

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users?_limit=2',
  method: 'GET'
};

let request = https.request(options, (res) => {
...
```

نلاحظ أنه وبدلًا من تمرير المسار كاملًا فإننا نمرره على قسمين ضمن الخاصيتين `host` و `path` حيث تُعبّر الخاصية `host` عن عنوان النطاق أو عنوان IP للخادم الوجهة، أما الخاصية `path` فهي كل ما يلي بعد ذلك ضمن المسار بما فيها معاملات الاستعلام query parameters التي تأتي بعد إشارة الاستفهام.

ويمكن أن تحتوي الخيارات على بيانات مفيدة أخرى للطلب المرسل مثل الترويسات المرسلة وهي بيانات وصفية عن الطلب نفسه، فمثلًا عادة تتطلب الواجهة البرمجية API تحديد صيغة البيانات المرسلة من عدة صيغ مدعومة مثل JSON أو CSV أو XML، ولتحديد الصيغة التي يطلبها المستخدم يمكن للواجهة البرمجية معاينة قيمة الترويسة `Accept` ضمن الطلب الوارد إليها وتحدد على أساسه الصيغة المناسبة لإرسالها.

تُعبّر الترويسة `Accept` عن نوع البيانات التي يمكن للمستخدم التعامل معها، وبما أننا نتعامل مع واجهة برمجية تدعم صيغة JSON فقط، فيمكننا إضافة الترويسة `Accept` وإضافة قيمة لها توضح أننا نريد البيانات بصيغة JSON كالتالي:

```
const https = require('https');

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users?_limit=2',
  method: 'GET',
  headers: {
    'Accept': 'application/json'
  }
};
```

وبذلك نكون قد تعرفنا على أكثر أربعة خيارات استخدامًا ضمن طلبات HTTP وهي عنـوان المضـيف `host` و المسار `path` ونوع الطلب `method` والترويسات `headers`، ويوجد العديـد من الخيـارات الأخـرى المدعومـة يمكنك الرجوع إلى التوثيق الرسمي لها ضمن نود للتعرف عليها.

والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج لنختبر طريقة إرسال الطلبات بتمرير كائن الخيارات فقط:

```
$ node getRequestWithRequest.js
```

ليظهر لنا بيانات الرد مطابقة للأمثلة السابقة:

```
Retrieved all data
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
    phone: '1-770-736-8031 x56442',
    website: 'hildegard.org',
    company: {
      name: 'Romaguera-Crona',
      catchPhrase: 'Multi-layered client-server neural-net',
      bs: 'harness real-time e-markets'
    }
  },
  {
    id: 2,
    name: 'Ervin Howell',
    username: 'Antonette',
    email: 'Shanna@melissa.tv',
    address: {
```

```
        street: 'Victor Plains',
        suite: 'Suite 879',
        city: 'Wisokyburgh',
        zipcode: '90566-7771',
        geo: [Object]
      },
      phone: '010-692-6593 x09125',
      website: 'anastasia.net',
      company: {
        name: 'Deckow-Crist',
        catchPhrase: 'Proactive didactic contingency',
        bs: 'synergize scalable supply-chains'
      }
    }
]
```

تختلف متطلبات الواجهات البرمجية API بحسب الجهة المطورة لها، لـذا من الضـروري التعامـل مـع كـائن الخيارات `options` لتخصـيص الطلب بحسـب حاجـة التطـبيق والخـادم، من تحديـد نـوع البيانـات المطلوبـة وإضافة الترويسات المناسبة وبعض التخصيصات الأخرى.

أرسلنا ضمن كل الأمثلة السابقة طلبـات فقـط من نـوع `GET` لجلب البيانـات، وفي الفقـرة التاليـة سـنتعلم طريقة إرسال الطلبات من نوع `POST` والتي تستخدم لرفع البيانات إلى الخادم.

13.4 إرسال طلب من نوع POST

نستخدم الطلبات من نوع `POST` لرفع البيانات إلى الخادم أو لطلب إنشاء بيانـات جديـدة من قبـل الخـادم، وفي هذه الفقرة سنتعرف على طريقة إرسال مثل هذه الطلبات في نود عبر إرسال طلب إنشـاء مسـتخدم جديـد إلى المسار `users` على الواجهة البرمجية API.

يمكننا إعادة استخدام بعض الشيفرات من مثال إرسال طلب من نوع `GET` السابق لإرسال طلبات من نوع `POST` مع إجراء بعض التعديلات عليها:

- تعديل نوع الطلب ضمن كائن الخيارات `options` ليصبح `POST`.
- تعيين ترويسة نوع المحتوى المُرسل وهو في حالتنا بصيغة JSON.
- التأكد من رمز الحالة للرد لتأكيد نجاح إنشاء مستخدم جديد.
- رفع بيانات المستخدم الجديد.

نبدأ بإنشاء ملف جافاسكربت جديد بالاسم `postRequest.js` ونفتحه ضمن محرر النصوص:

```
$ nano postRequest.js
```

ونبدأ كما سابقًا باستيراد الوحدة `https` وتعريف كائن الخيارات `options` كالتالي:

```
const https = require('https');

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users',
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json; charset=UTF-8'
  }
};
```

ونعدل خيار مسار الطلب `path` ليتوافق مع مسار إرسال الطلبات من نوع `POST` على الخادم، ونعدل خيار نوع الطلب المرسل `method` إلى القيمة `POST`، وأخيرًا نضيف الترويسة `Content-Type` ضمن الخيارات والتي تدل الخادم على نوع البيانات التي أرسلناها مع الطلب وهي في حالتنا بصيغة JSON وبترميز من نوع UTF-8، ثم نُرسل الطلب باستدعاء التابع `request()` كما فعلنا تمامًا عند إرسال طلب من نوع `GET` سابقًا ولكن هذه المرة سنتحقق من رمز الحالة للرد بقيمة تختلف عن 200 كالتالي:

```
...
const request = https.request(options, (res) => {
  if (res.statusCode !== 201) {
    console.error(`Did not get a Created from the server. Code: $
{res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });
```

```
  res.on('close', () => {
    console.log('Added new user');
    console.log(JSON.parse(data));
  });
});
```

تحققنا من صحة العملية بالتحقق من قيمة رمز الحالة بأن يساوي 201، وهـو الرمـز الـذي يـدل على إنشـاء مورد جديد على الخادم بنجاح، ويُعبّر الطلب المرسل عن إنشاء مستخدم جديد لهـذا سـنحتاج لرفـع بيانـات هـذا المستخدم وإرفاقها ضمن الطلب، لذا سننشئ تلك البيانات كالتالي:

```
...

const requestData = { name: '
  New User', username: 'Dan',
  email: 'user@learn.com',
  address: {
    street: 'North Pole',
    city: 'Murmansk', zipcode
    : '12345-6789',
  },
  phone: '555-1212',
  website: 'learn.com',
  company: {
    name: 'learn',

    catchPhrase: 'Welcome to Learn', bs: '
    cloud scale security'
  }
};

request.write(JSON.stringify(requestData));
```

عرفنا بيانات المستخدم الجديد ضمن المتغير `requestData` على شـكل كـائن جافاسـكربت يحتـوي على بيانات المستخدم، ونلاحظ أننا لم نرفق قيمة المعرف `id` للمستخدم حيث أن هذه القيمة يولدها الخـادم تلقائيًـا للبيانات الجديدة، ثم استدعينا التابع `request.write()` والذي يقبل سلسـلة نصـية أو كـائن مخـزن مـؤقت

buffer ليتم إرسالها ضمن الطلب، وبما أن البيانات لدينا ضمن المتغير `requestData` هي كائن جافاسكربت فيجب تحويله إلى سلسلة نصية باستخدام `JSON.stringify`.

ولإنهاء عملية الإرسال ننهي الطلب عبر استدعاء `request.end()` ونتحقق من حدوث أي أخطاء في عملية الإرسال كالتالي:

```
...

request.end();

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a request: ${err.message}`);
});
```

من الضروري استدعاء التابع `end()` لإنهاء الطلب وللإشارة إلى نود بأن كل البيانات التي نريد إرسالها ضمن الطلب قد أرفقت وأصبح بالإمكان إرساله.

والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج ونتأكد من عملية إنشاء المستخدم الجديد:

```
$ node postRequest.js
```

سنحصل على الخرج التالي:

```
Added new user
{
  name: 'New User',
  username: 'learn',
  email: 'user@learn.com',
  address: { street: 'North Pole', city: 'Murmansk', zipcode: '12345-6789' },
  phone: '555-1212',
  website: 'learn.com
  ', company: {
    name: 'learn',
    catchPhrase: 'Welcome to the learn', bs: '
    cloud scale security'
  },
  id: 11
```

```
}
```

ما يعني أن الطلب تم بنجاح، حيث أعاد الخادم بيانات المستخدم التي أرسلناها مضافًا إليها قيمة معرّف المستخدم ID التي تم توليدها له، وبذلك نكون قد تعلمنا طريقة إرسال الطلبات من نوع POST لرفع البيانات إلى الخادم باستخدام نود، وفي الفقرة التالية سنتعلم طريقة إرسال الطلبات من نوع PUT للتعديل على بيانات موجودة مسبقًا.

13.5 إرسال طلب من نوع PUT

تُستخدم الطلبات من نوع PUT لرفع البيانات إلى الخادم بشكل مشابه للطلبات من نوع POST، ولكن الفرق أنه عند تنفيذ طلب من نوع PUT عدة مرات سنحصل على نفس النتيجة، بينما عند تكرار نفس طلب POST عدة مرات سنضيف بذلك البيانات المرسلة أكثر من مرة إلى الخادم، وطريقة إرسال هذا الطلب مشابهة للطلب من نوع POST حيث نعرف الخيارات ونُنشئ الطلب ونكتب البيانات التي نريد رفعها إلى الطلب ثم نتحقق من الرد الوارد في نتيجة الطلب.

نختبر ذلك بإنشاء طلب من نوع PUT لتعديل اسم المستخدم لأول مستخدم، وبما أن طريقة إرسال الطلب مشابهة لطريقة إرسال الطلب من نوع POST يمكننا الاستفادة من المثال السابق ونسخ الملف postRequest.js إلى ملف جديد بالاسم putRequest.js كالتالي:

```
$ cp postRequest.js putRequest.js
```

نفتح الملف putRequest.js ضمن محرر النصوص:

```
$ nano putRequest.js
```

ونعدل نوع الطلب إلى PUT ومساره إلى https://jsonplaceholder.typicode.com/users/1 والبيانات المرسلة كالتالي:

```
const https = require('https');

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users/1',
  method: 'PUT',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json; charset=UTF-8'
```

```
  }
};

const request = https.request(options, (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: $
{res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Updated data');
    console.log(JSON.parse(data));
  });
});

const requestData = {
  username: 'learn'
};

request.write(JSON.stringify(requestData));

request.end();

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a request: $
{err.message}`);
});
```

نلاحظ تعديل قيم المسار `path` ونوع الطلب `method` ضمن كائن الخصائص `options` حيث يحوي المسار على معرّف المستخدم الذي نود تعديل بياناته، ثم نتحقق من من رمز الحالة للطلب بأن يكون بالقيمة 200 ما يدل على نجاح الطلب، ونلاحظ أن البيانات التي أرسلناها تحوي فقط على الخصائص التي نريد تحديثها من بيانات المستخدم.

والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج:

```
$ node putRequest.js
```

ليظهر الخرج التالي:

```
Updated data
{ username: 'learn', id: 1 }
```

أرسلنا بنجاح طلب من نوع `PUT` لتعديل بيانات مستخدم موجودة مسبقًا على الخادم، وبذلك نكون قد تعلمنا طرق طلب البيانات ورفعها وتحديثها، وسنتعلم في الفقرة التالية كيف يمكن حذف البيانات من الخادم بإرسال طلب من النوع `DELETE`.

13.6 إرسال طلب من نوع DELETE

تستخدم الطلبات من نوع `DELETE` لحذف البيانات من الخادم، ويمكن أن يحتوي الطلب على بيانات مرفقة ضمنه ولكن معظم الواجهات البرمجية API لا تتطلب ذلك، حيث يستخدم هذا النوع من الطلبات لحذف بيانات كائن ما كليًا من الخادم.

سنرسل في هذه الفقرة طلب من هذا النوع لحذف بيانات أحد المستخدمين، وطريقة إرسال هذا الطلب مشابهة لطريقة إرسال طلب من نوع `GET`، لذا يمكننا نسخ ملف المثال السابق `getRequestWithRequest.js` إلى ملف جديد بالاسم `deleteRequest.js` كالتالي:

```
$ cp getRequestWithRequest.js deleteRequest.js
```

ونفتح الملف الجديد ضمن محرر النصوص:

```
$ nano deleteRequest.js
```

ونعدل الشيفرة لإرسال طلب حذف لأول مستخدم كالتالي:

```
const https = require('https');

const options = {
```

```
  host: 'jsonplaceholder.typicode.com',
  path: '/users/1',
  method: 'DELETE',
  headers: {
    'Accept': 'application/json',
  }
};

const request = https.request(options, (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: $
{res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Deleted user');
    console.log(JSON.parse(data));
  });
});

request.end();

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a request: $
{err.message}`);
});
```

عدلنا قيمة المسار `path` ضمن كائن خيارات الطلب ليحوي معرّف المستخدم الذي نريد حذفه، وعدلنا نـوع الطلب إلى `DELETE`، والآن نحفظ الملف ونخرج منه ثم ننفذ البرنامج كالتالي:

```
$ node deleteRequest.js
```

لنحصل على الخرج:

```
Deleted user
{}
```

لا تعيد الواجهة البرمجية أي بيانات ضمن جسم الرد الوارد، ولكن رمز الحالة لهذا الرد يكون 200 أي تم حذف بيانات المستخدم بنجاح، وبذلك نكون قد تعرفنا على طريقة إرسال طلبات من نوع `DELETE` أيضًا في نود.

13.7 خاتمة

تعرفنا في هـذا الفصـل على طريقـة إرسـال الطلبـات في نـود بأنواعهـا مختلفـة مثـل `GET` و `POST` و `PUT` و `DELETE` دون استخدام أي مكتبات خارجية وفقط باستخدام الوحدة البرمجية `https` التي يوفرها نود، وتعرفنا على طريقة خاصة لإرسال الطلبات من نوع `GET` باستخدام التابع `get()` وكيف أن باقي الطلبات يمكن إرسـالها باستخدام التابع `request()`، وتعاملنا ضمن الأمثلة مع واجهة برمجية API عامة ويمكن بنفس الطريقة إرسـال الطلبات إلى مختلف أنواع الواجهات البرمجية.