TD: Débogage de NodeJS

Objectifs

- Diagnostiquer les problèmes avec Chrome DevTools
- “Logging” avec Node.js
- Activation des journaux de débogage
- Activation des journaux de débogage “core” Node.js
- Augmentation de la taille de la trace de la pile (stack trace)

Diagnostiquer les problèmes avec Chrome DevTools

Node.js expose un utilitaire de débogage via l'indicateur de processus **--inspect**, qui nous permet de déboguer et de profiler nos processus Node.js à l'aide de l'interface **Chrome DevTools**.
L'intégration est activée via le protocole Chrome DevTools. L'existence de ce protocole signifie que des outils peuvent être écrits pour s'intégrer à Chrome DevTools.

Dans cet exemple, nous allons apprendre à utiliser Chrome DevTools pour diagnostiquer les problèmes dans une application Web.

Tout d'abord, configurons un répertoire et les fichiers requis pour cet exemple:

$ mkdir td13 $

cd td13

$ npm init --yes

$ npm install express --save

$ touch server.js random.js

Ajoutez le code source suivant à **server.js** pour créer notre serveur Web:

```
const express = require("express");
const app = express();
const random = require("./random");
app.get("/:number", (req, res) => {
      const number = req.params.number;
      res.send(random(number).toString());
});
app.listen(3000, () => {
      console.log("Le serveur ecoute sur le port 3000");
});
```

Ajoutez le code source suivant à **random.js**. Ce sera un module avec lequel nous interagissons via notre serveur:

```
module.exports = (n) => {
      const randomNumber = Math.floor(Math.random() * n) + "1";
      return randomNumber;
};
```

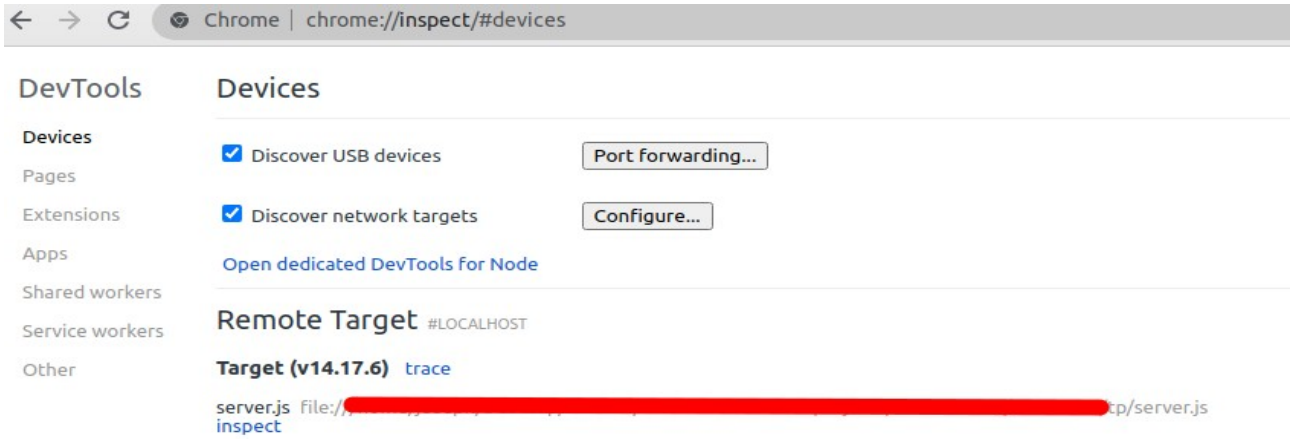
Nous avons maintenant une application prête à déboguer. Dans cet exemple, nous allons utiliser **Chrome DevTools** pour déboguer une route dans notre application. Notre application répondra avec un nombre aléatoire entre 0 et le nombre que nous spécifions dans la route. Par exemple, http://localhost:3000/10 doit renvoyer un nombre aléatoire compris entre 1 et 10:

Démarrez le programme avec **$ node server.js** et accédez à http://localhost:3000/10. Actualisez le point de terminaison plusieurs fois et vous remarquerez que le programme répondra souvent avec un nombre supérieur à 10. Cela semble erroné, alors déboguons pour essayer de comprendre pourquoi.

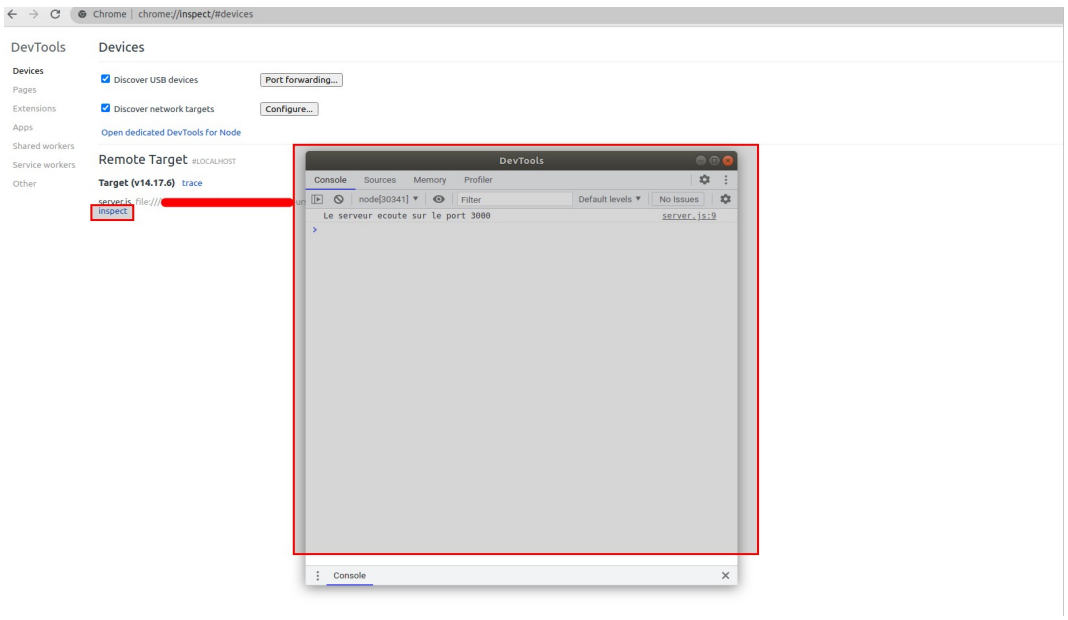
Tout d'abord, nous devons démarrer notre programme avec le débogueur activé. Pour ce faire, nous devons transmettre l'argument **--inspect** à notre processus Node.js:

```
$ node --inspect server.js
```

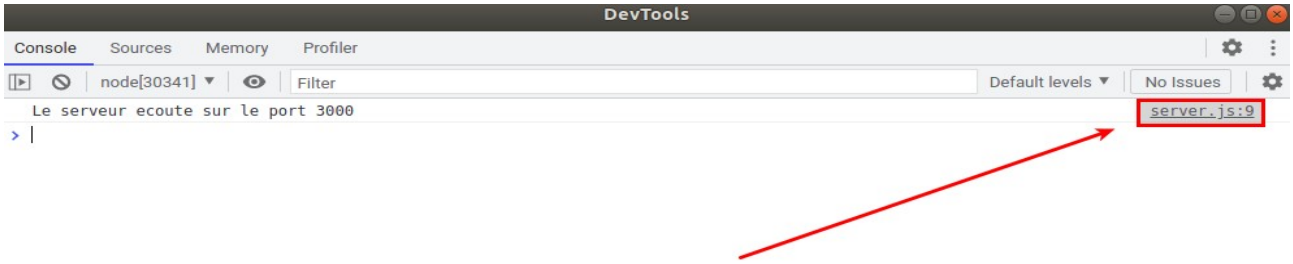
Accédez à chrome://inspect/#devices dans Google Chrome. Attendez-vous à voir le résultat suivant:

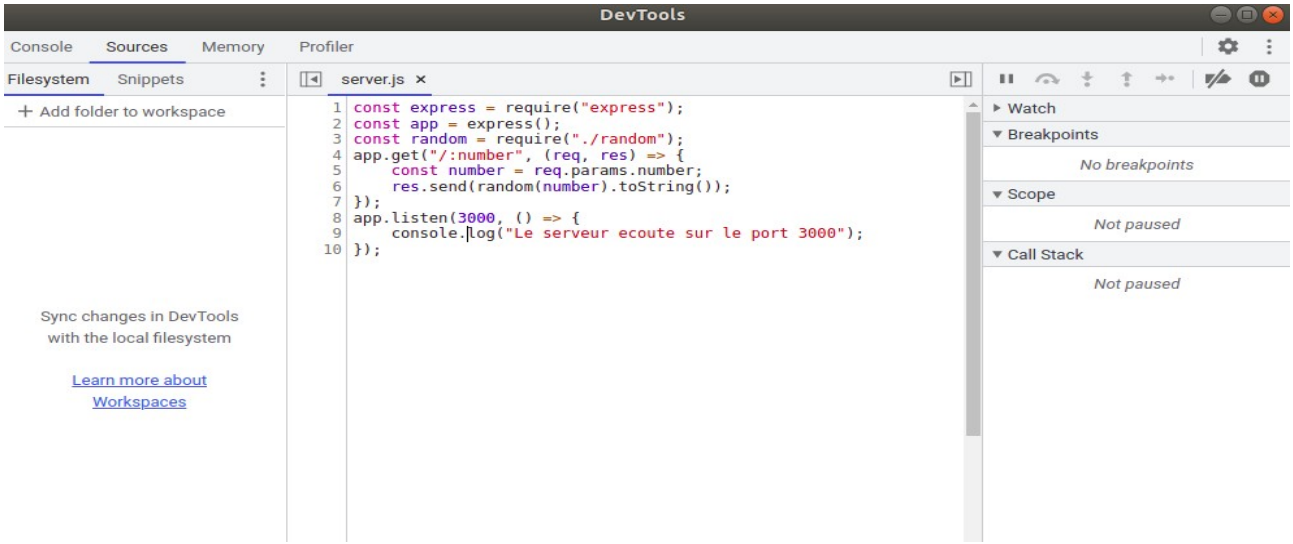


Observez que notre **server.js** apparaît en tant que “Remote Target”. Cliquez sur le lien “**inspect**” et la fenêtre Chrome DevTools s'ouvrira, comme illustré dans l'image suivante:

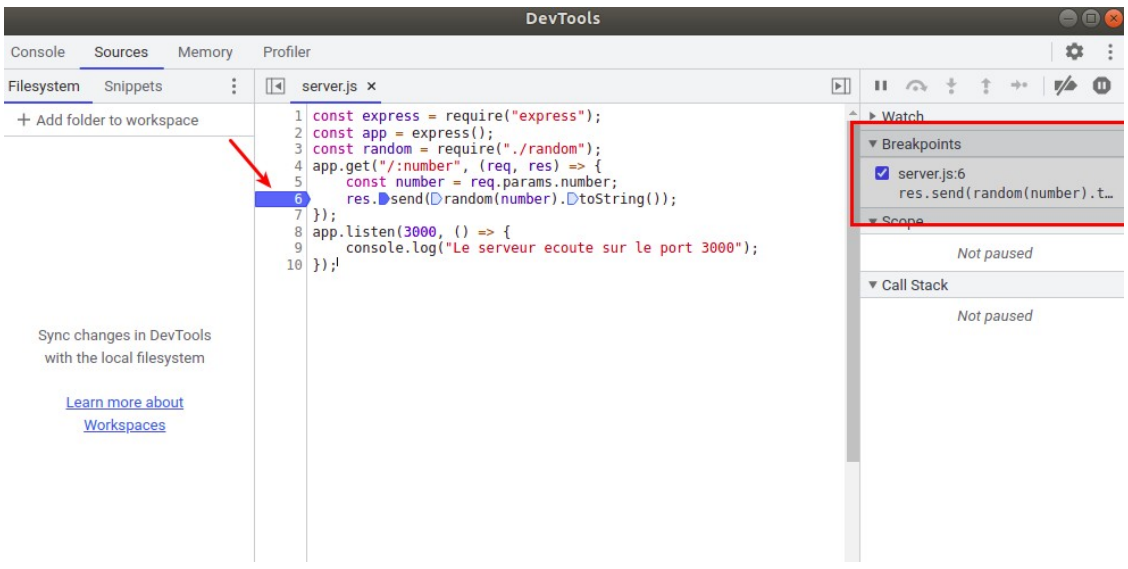


Cliquez sur “server.js:x”. Cela devrait garantir que notre fichier **server.js** est ouvert:



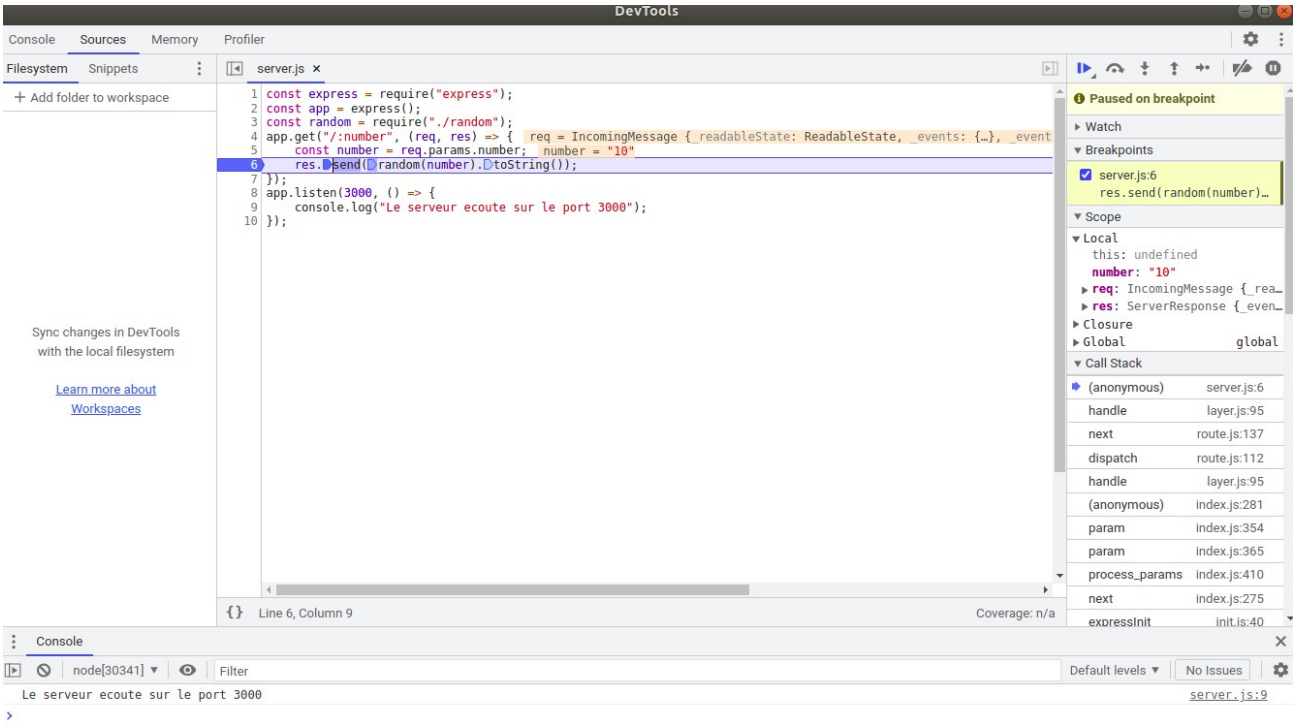


Maintenant, nous pouvons ajouter un point d'arrêt (Breakpoint). Cliquez sur le chiffre 6 dans la colonne “Ligne de code” à gauche de notre code. Un petit cercle rouge (ou un rectangle bleu) devrait apparaître à côté du numéro. Si vous cliquez sur “Show Debugger” dans le coin supérieur droit, vous devriez voir le point d'arrêt répertorié dans le volet “Breakpoints pane”. L'interface Chrome DevTools doit ressembler à ceci:

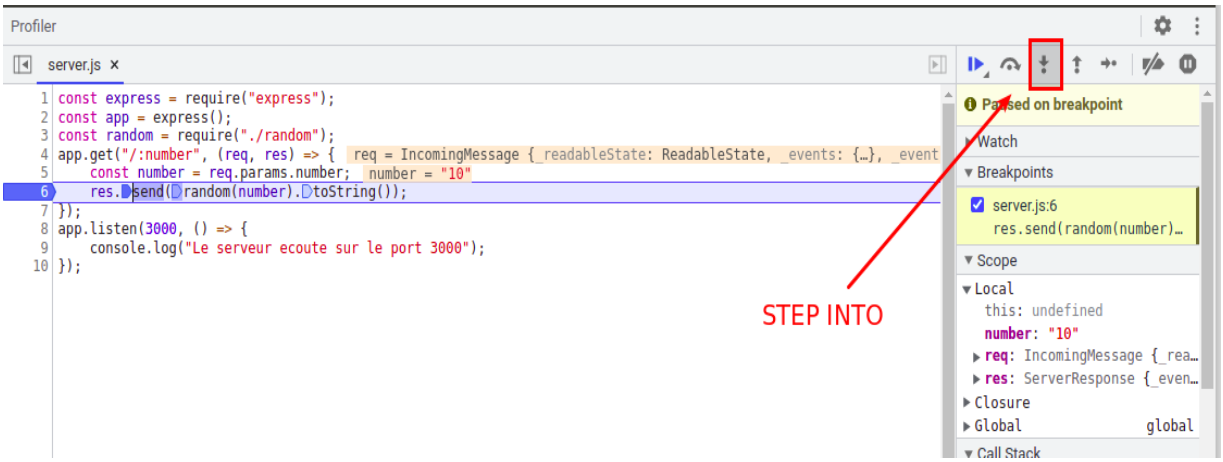


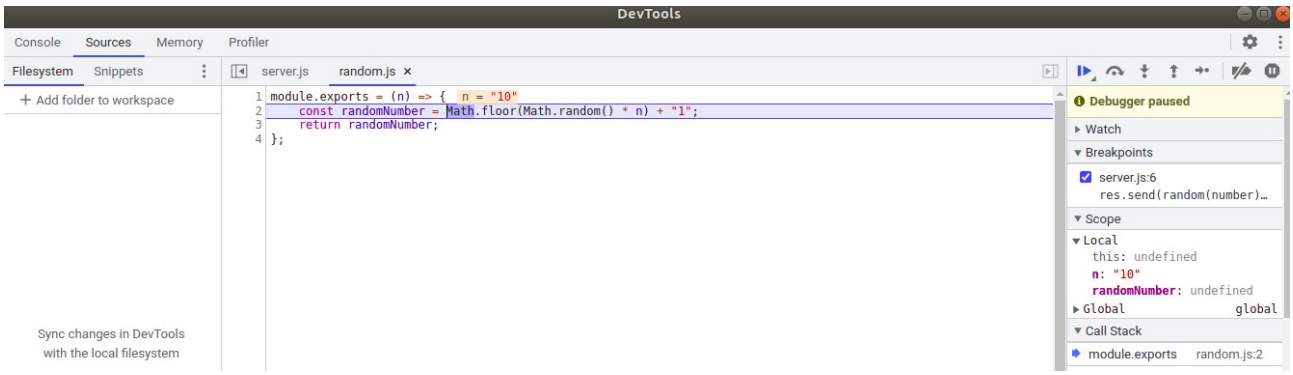
Maintenant, ouvrons une nouvelle fenêtre de navigateur standard et naviguons jusqu'à http://localhost:3000/10. La requête se bloquera car elle a atteint le point d'arrêt que nous avons enregistré à la ligne 6.

Revenez à Chrome DevTools. Vous devriez remarquer qu'il y a une info-bulle indiquant Pause sur le point d'arrêt (Pause on breakpoint) en haut à droite de l'interface. De plus, à droite de l'interface, vous devriez voir un panneau Call Stack, qui détaille les trames d'appel:

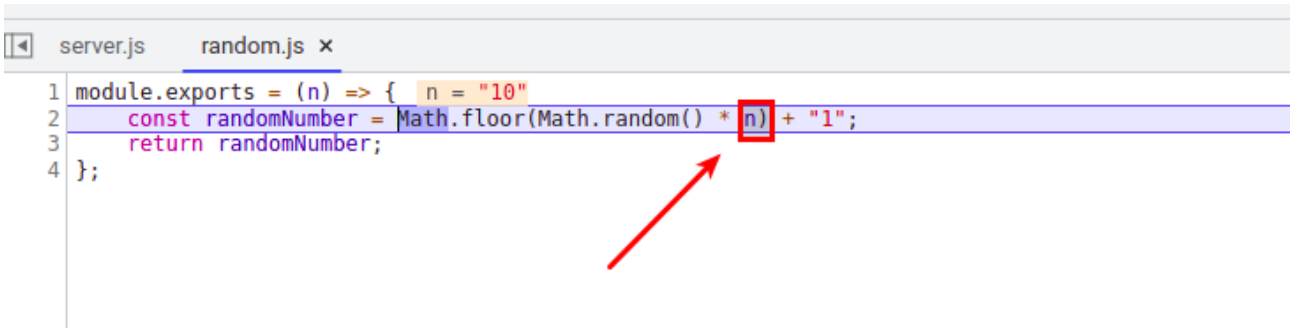


Le débogueur attend que nous intervenions. Nous pouvons choisir d'entrer ou de sortir de l'instruction suivante. Entrons dans la fonction. Pour ce faire, cliquez sur l'icône d'une flèche pointant vers un cercle (ces icônes se trouvent juste au-dessus du message “Paused on breakpoint”). Lorsque vous survolez chaque icône, une info-bulle apparaîtra décrivant le comportement de l'icône. Une fois que vous êtes intervenu, vous verrez que nous sommes passés à notre fichier **random.js**:

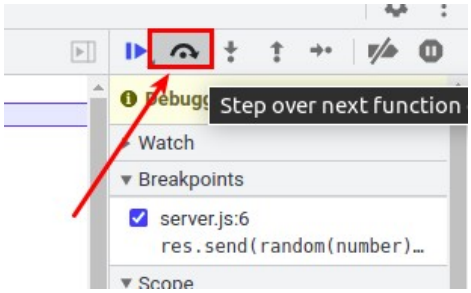


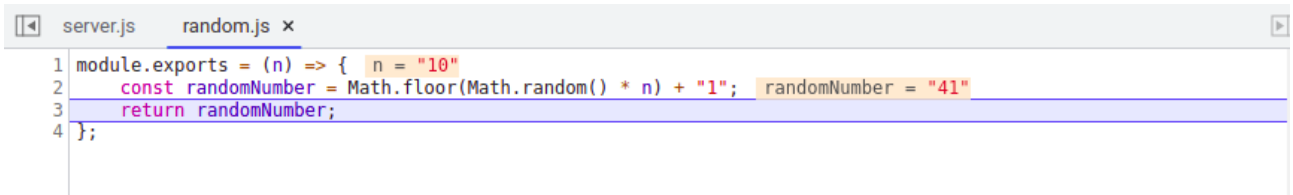


Pendant que nous sommes dans **random.js**, nous pouvons survoler les valeurs pour vérifier si elles correspondent à ce que nous attendons d'elles. Nous pouvons voir que n = 10, comme prévu.



Passez par-dessus (step-over) la fonction (en cliquant sur la flèche semi-circulaire avec un point en dessous), puis inspectez la valeur de **randomNumber**. Dans la capture d'écran, le nombre est 41, ce qui est supérieur à 10. Cela nous aide à déterminer que l'erreur est dans notre logique **randomNumber** de la ligne précédente.





Maintenant que nous avons identifié la ligne sur laquelle se trouve l'erreur, il est plus facile de localiser l'erreur. Nous ajoutons la chaîne "1" plutôt que le numéro 1. Remplacer "1" par 1:

```
const randomNumber = Math.floor(Math.random() * n) + 1;
```

La possibilité de déboguer les applications Node.js est fournie par le moteur JavaScript V8. Lorsque nous passons au processus de Node l'argument **--inspect**, le processus Node.js commence à écouter un client de débogage. Plus précisément, c'est l'inspecteur V8 qui ouvre un port qui accepte les connexions WebSocket. La connexion WebSocket permet au client et à l'inspecteur V8 d'interagir.

Dans cet exemple, nous définissons un point d'arrêt dans la fenêtre Chrome DevTools. Lorsque la ligne de code sur laquelle le point d'arrêt est enregistré est rencontrée, la boucle d'événement “Event loop” (thread JavaScript) sera interrompue. L'inspecteur V8 enverra alors un message au client via la connexion WebSocket. Le message de l'inspecteur V8 détaille la position et l'état du programme. Le client peut mettre à jour son état en fonction des informations qu'il reçoit.

De même, si l'utilisateur choisit d'entrer dans une fonction, une commande est envoyée à l'inspecteur V8 pour lui demander de reprendre temporairement l'exécution du script, en le suspendant à nouveau par la suite. Comme auparavant, l'inspecteur V8 renvoie un message au client détaillant la nouvelle position et l'état.

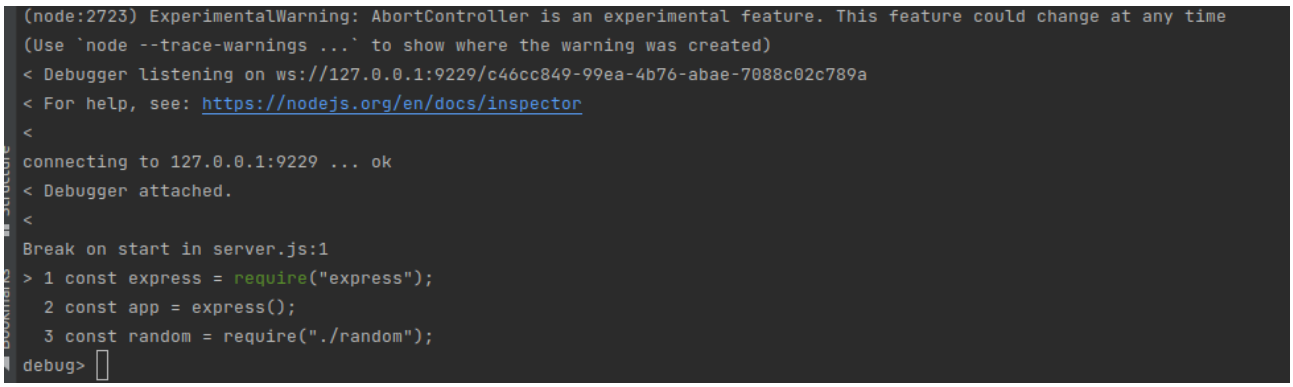
Node.js fournit également un indicateur que nous pouvons utiliser pour suspendre une application au démarrage. Cette fonctionnalité nous permet de configurer des points d'arrêt (breakpoints) avant que quoi que ce soit ne s'exécute. Cela peut également aider lors du débogage d'une erreur qui se produit pendant la phase de configuration de votre application.

Cette fonctionnalité peut être activée avec l'indicateur **--inspect-brk**. Voici comment démarrer **server.js** à l'aide de l'indicateur **--inspect-brk**:

```
$ node --inspect-brk server.js
```

Node.js fournit un inspecteur de ligne de commande, qui peut être utile lorsque nous n'avons pas accès à une interface utilisateur graphique. Nous pouvons exécuter l'application à partir de cet exemple à l'aide du débogueur en ligne de commande avec la commande suivante:

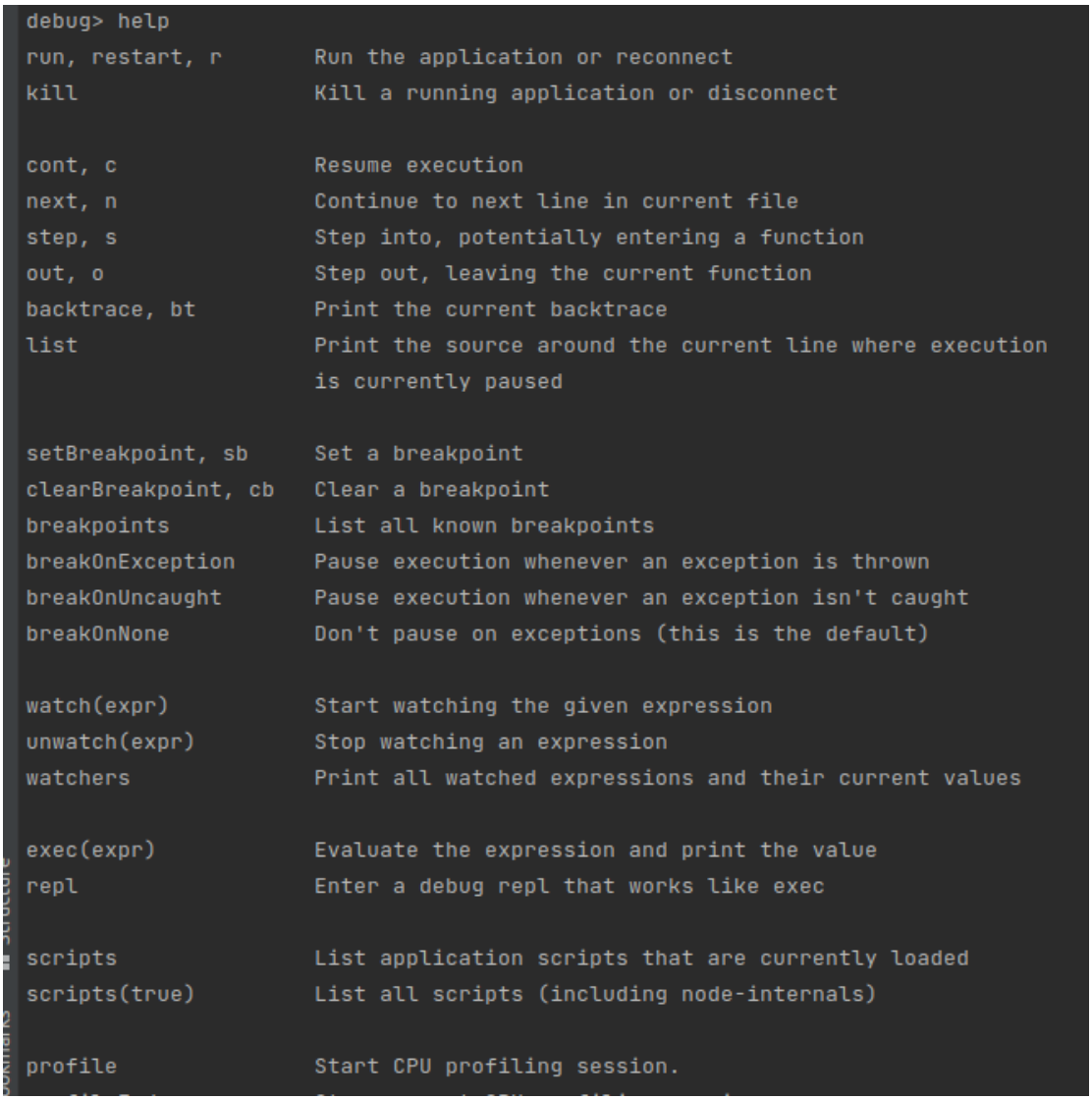
```
$ node inspect server.js
```



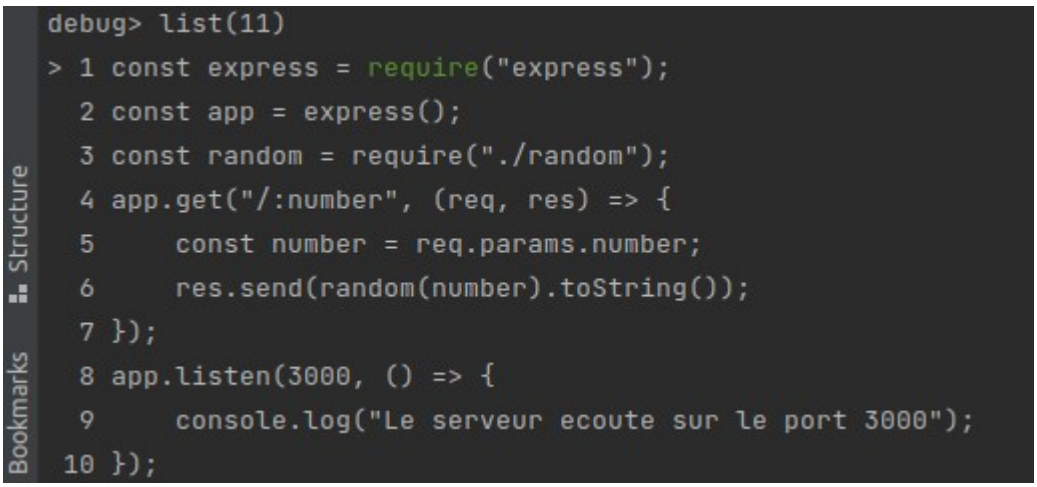
Cette commande nous amènera en mode débogage et affichera les trois premières lignes de **server.js**.

Le mode débogage mettra notre programme en pause à la première ligne.

Le mode de débogage fournit une série de commandes et de fonctions que nous pouvons utiliser pour parcourir et déboguer notre programme. Vous pouvez afficher la liste complète de ces commandes en tapant **help** et en appuyant sur Entrée.

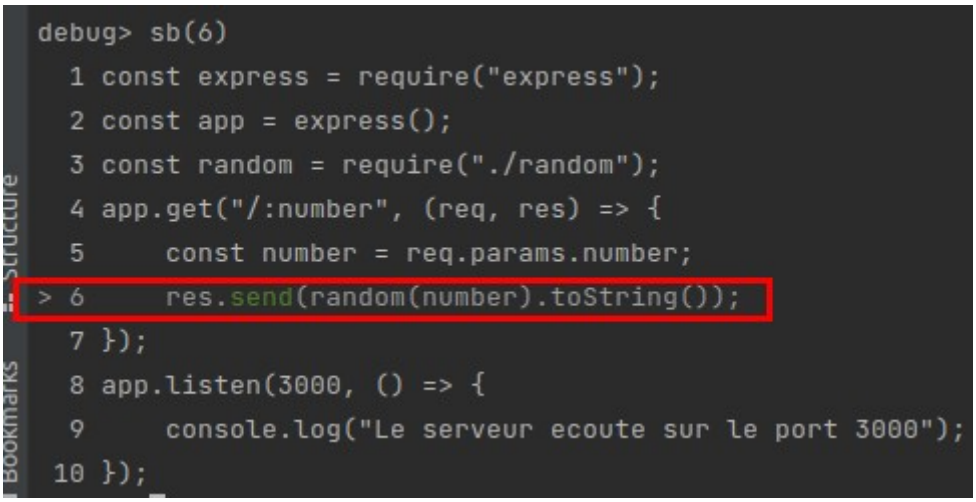


L'une des fonctions est la fonction **list()**, qui listera un nombre spécifié de lignes suivantes. Par exemple, nous pouvons taper **list(11)** pour afficher les 12 lignes de notre programme:



Nous pouvons utiliser la fonction **setBreakpoint()** pour définir un point d'arrêt. Nous devons fournir à cette fonction le numéro de ligne sur lequel nous souhaitons définir le point d'arrêt. Il existe également un raccourci pour cette fonction: **sb()**.

Définissons un point d'arrêt sur la ligne 6 en tapant setBreakpoint(6) ou sb(6):



Le caret (>) indique qu'un point d'arrêt a été défini sur la ligne 6.

Le programme est toujours en pause. Nous pouvons demander au processus de commencer à s'exécuter en tapant la commande continue, **cont** ou **c.**

```
debug> cont
< Le serveur ecoute sur le port 3000
<
```

Envoyons une requête en utilisant **cURL** dans une nouvelle fenêtre de terminal:

$ curl http://localhost:3000/10

La commande se bloquera, car elle a atteint notre point d'arrêt sur la ligne 6 de **server.js.** Si nous revenons à la session de débogage, nous verrons que le débogueur a détecté qu'un point d'arrêt a été atteint:

```
break in server.js:6
  4 app.get("/:number", (req, res) => {
  5     const number = req.params.number;
> 6     res.send(random(number).toString());
  7 });
  8 app.listen(3000, () => {
debug>
```

Maintenant, pour entrer (step into) dans la fonction, nous tapons la commande **step**:

```
debug> step
break in random.js:2
  1 module.exports = (n) => {
> 2     const randomNumber = Math.floor(Math.random() * n) + 1;
  3     return randomNumber;
  4 };
```

Cela va dans le fichier **random.js**. Notez que l'utilitaire de débogage en ligne de commande fournit une interface similaire à Chrome DevTools, mais sans interface utilisateur graphique.

Nous pouvons imprimer les références dans la portée actuelle à l'aide de la commande **exec**. Tapez **exec n** pour afficher la valeur de **n**:

```
debug> exec n
'10'
```

Nous pouvons maintenant sortir en utilisant la commande **out**. Cela nous ramènera dans notre fichier **server.js**:

```
debug> out
break in server.js:6
  4 app.get("/:number", (req, res) => {
  5     const number = req.params.number;
> 6     res.send(random(number).toString());
  7 });
  8 app.listen(3000, () => {
```

Pour quitter le débogueur, vous pouvez taper **.exit** ou entrer **Ctrl + C** deux fois.

“Logging” avec Node.js

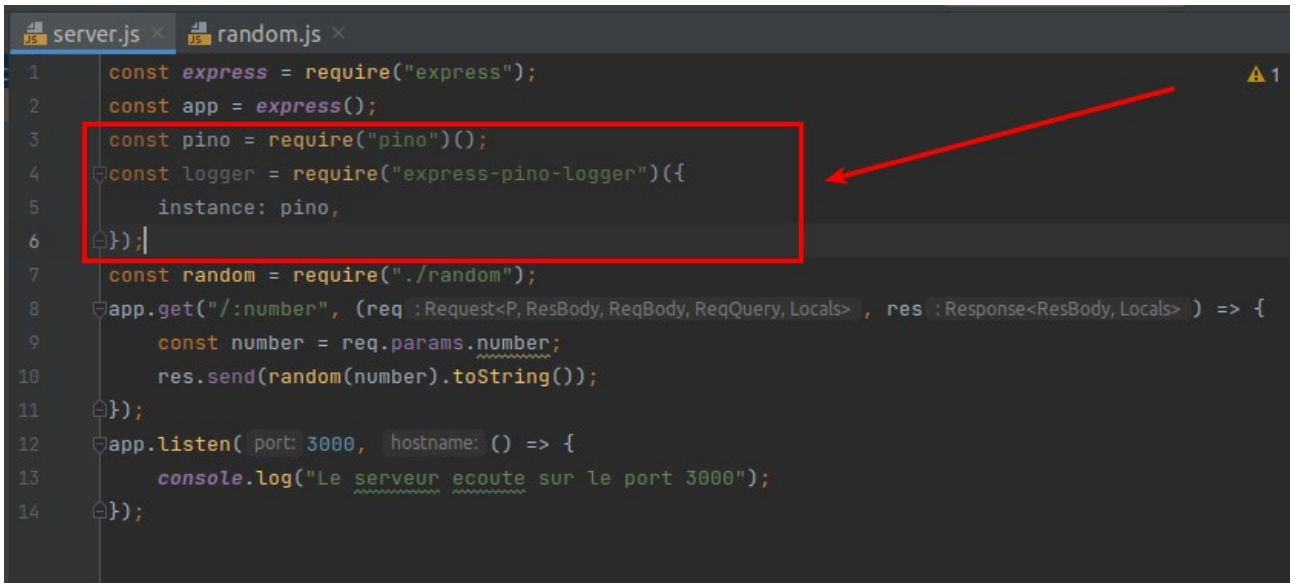
Une journalisation (logging) efficace peut vous aider à comprendre ce qui se passe dans une application. Les journaux (logs) peuvent aider à trier les causes des plantages ou des échecs rétrospectivement en vous aidant à voir ce qui se passait dans votre application avant le plantage ou l'échec.

La journalisation peut également être utilisée pour faciliter la collecte de données. Par exemple, si vous enregistrez tous les accès aux points de terminaison sur votre application Web, vous pouvez rassembler tous les journaux de requêtes pour déterminer quels sont les points de terminaison les plus visités.

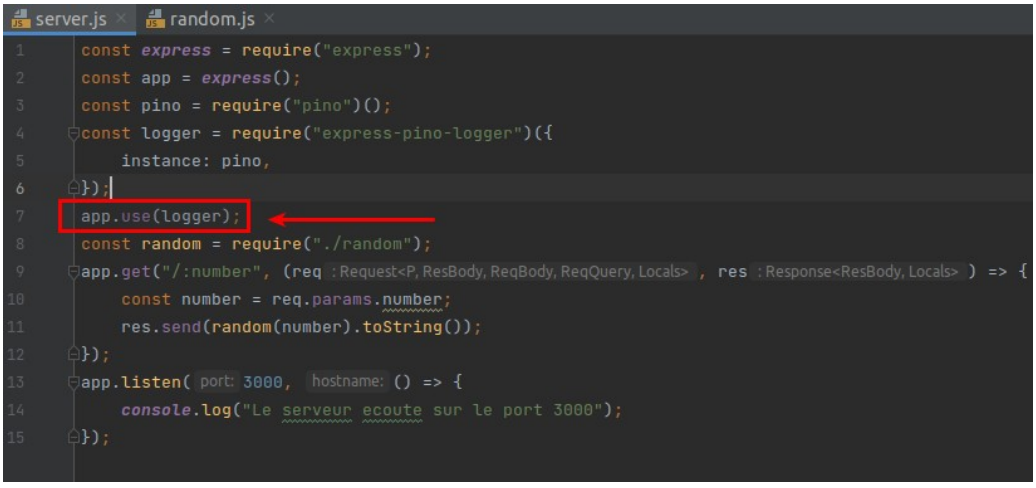
Dans cet exemple, nous examinerons la journalisation avec “Pino”, un enregistreur basé sur JSON. Nous allons utiliser le module **express-pino-logger** pour ajouter la journalisation via un middleware Express.js. Tout d'abord, commencez par installer les modules **pino** et **express-pino-logger**:

```
$ npm install pino express-pino-logger
```

Maintenant, nous pouvons importer **pino** et **express-pino-logger** dans **server.js**.



Maintenant, nous pouvons enregistrer notre enregistreur **pino** en tant que middleware Express.js. Ajoutez la ligne suivante sous la déclaration de **logger** ajoutée à l'étape précédente:



Nous sommes prêts à ajouter un message de journal **Pino** à notre requête. Nous ajouterons un message de journal qui nous informe que nous générons un nombre aléatoire. Ajoutez la ligne suivante à votre fonction de gestionnaire de requêtes. Cela devrait être la première ligne de votre fonction de gestionnaire de requêtes:

```
app.use(logger);
const random = require("./random");
app.get("/:number", (req :Request<P, ResBody, ReqBody, ReqQuery, Locals> , res :Response<ResBody, Locals> ) => {
    req.log.info("Générer un nombre aléatoire");
    const number = req.params.number;
    res.send(random(number).toString());
});
app.listen( port: 3000, hostname: () => {
    console.log("Le serveur ecoute sur le port 3000");
});
```

Nous allons maintenant convertir notre instruction **console.log()** existante pour utiliser le **pino** logger à la place:

```
app.get("/:number", (req :Request<P, ResBody, ReqBody, ReqQuery, Locals> , res :Response<ResBody, Locals> ) => {
    req.log.info("Générer un nombre aléatoire");
    const number = req.params.number;
    res.send(random(number).toString());
});
app.listen( port: 3000, hostname: () => {
    pino.info("Le serveur ecoute sur le port 3000");
});
```

Nous pouvons démarrer notre serveur et observer que notre message de journal est affiché, au format JSON:

```
(base) joseph@joseph-HP-EliteBook:~[redacted]/tp$ node server.js
{"level":30,"time":1639356093545,"pid":6093,"hostname":"joseph-HP-EliteBook","msg":"Le serveur ecoute sur le port 3000"}
```

Accédez à http://localhost:3000/10 dans votre navigateur (ou envoyez une requête HTTP GET à l'aide de cURL). Attendez-vous à voir la sortie de journal suivante dans votre fenêtre de terminal:

```
Terminal:  Local  Local (2)  +
{"level":30,"time":1639356093545,"pid":6093,"hostname":"joseph-HP-EliteBook","msg":"Le serveur ecoute sur le port 3000"}
{"level":30,"time":1639356184059,"pid":6093,"hostname":"joseph-HP-EliteBook","req":{"id":1,"method":"GET","url":"/10","query":{},"params":{},"headers":{"host"
:"localhost:3000","connection":"keep-alive","sec-ch-ua":"\" Not A;Brand\";v=\"99\", \"Chromium\";v=\"96\", \"Google Chrome\";v=\"96\"","sec-ch-ua-mobile":"?0"
,"sec-ch-ua-platform":"\"Linux\"","upgrade-insecure-requests":"1","user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
96.0.4664.45 Safari/537.36","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exch
ange;v=b3;q=0.9","sec-fetch-site":"none","sec-fetch-mode":"navigate","sec-fetch-user":"?1","sec-fetch-dest":"document","accept-encoding":"gzip, deflate, br","
accept-language":"en-US,en;q=0.9,fr-FR;q=0.8,fr;q=0.7,ar-LB;q=0.6,ar;q=0.5","cookie":"Phpstorm-765b08c7=43941c52-b30c-4f77-8c00-ad2d1fd46856; Phpstorm-765b08c
9=aad16ba7-2a86-4163-acab-0b7e3e830cd7; Webstorm-6c2b712d=264135ad-20fc-4499-adeb-1848af7a3c65; connect.sid=s%3Alkk0ag7S0VF6nQg9AiHYRzONTe-VQ675.Co%2Brr0FvCWv
MAhEvr1iDlSFcJkX3Q67tp5d7NR5qe%2BE; SESSIONID=s%3AjD-sbDcTAGaF8NUB6tmNz0i50XMJuv6W.ZINajfxFnxhjG%2BaYK2LLLtvPl%2FjRny9XwdEDfyo8elc","if-none-match":"W/\"1-2ku
SN7rMzfGcB2DKt67EqDWQELA\""} "remoteAddress":"::1","remotePort":35194},"msg":"Générer un nombre aléatoire"
{"level":30,"time":1639356184081,"pid":6093,"hostname":"joseph-HP-EliteBook","req":{"id":1,"method":"GET","url":"/10","query":{},"params":{},"headers":{"host"
:"localhost:3000","connection":"keep-alive","sec-ch-ua":"\" Not A;Brand\";v=\"99\", \"Chromium\";v=\"96\", \"Google Chrome\";v=\"96\"","sec-ch-ua-mobile":"?0"
,"sec-ch-ua-platform":"\"Linux\"","upgrade-insecure-requests":"1","user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
96.0.4664.45 Safari/537.36","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exch
ange;v=b3;q=0.9","sec-fetch-site":"none","sec-fetch-mode":"navigate","sec-fetch-user":"?1","sec-fetch-dest":"document","accept-encoding":"gzip, deflate, br","
accept-language":"en-US,en;q=0.9,fr-FR;q=0.8,fr;q=0.7,ar-LB;q=0.6,ar;q=0.5","cookie":"Phpstorm-765b08c7=43941c52-b30c-4f77-8c00-ad2d1fd46856; Phpstorm-765b08c
9=aad16ba7-2a86-4163-acab-0b7e3e830cd7; Webstorm-6c2b712d=264135ad-20fc-4499-adeb-1848af7a3c65; connect.sid=s%3Alkk0ag7S0VF6nQg9AiHYRzONTe-VQ675.Co%2Brr0FvCWv
MAhEvr1iDlSFcJkX3Q67tp5d7NR5qe%2BE; SESSIONID=s%3AjD-sbDcTAGaF8NUB6tmNz0i50XMJuv6W.ZINajfxFnxhjG%2BaYK2LLLtvPl%2FjRny9XwdEDfyo8elc","if-none-match":"W/\"1-2ku
SN7rMzfGcB2DKt67EqDWQELA\""},"remoteAddress":"::1","remotePort":35194},"res":{"statusCode":200,"headers":{"x-powered-by":"Express","content-type":"text/html;
charset=utf-8","content-length":"1","etag":"W/\"1-wd/ZbuqMwrYnhSdbyjisJhJW4ng\""}},"responseTime":23,"msg":"request completed"}
{"level":30,"time":1639356184248,"pid":6093,"hostname":"joseph-HP-EliteBook","req":{"id":2,"method":"GET","url":"/favicon.ico","query":{},"params":{},"headers
":{"host":"localhost:3000","connection":"keep-alive","sec-ch-ua":"\" Not A;Brand\";v=\"99\", \"Chromium\";v=\"96\", \"Google Chrome\";v=\"96\"","sec-ch-ua-mob
ile":"?0","user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.45 Safari/537.36","sec-ch-ua-platform":"\"Linu
x\"","accept":"image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8","sec-fetch-site":"same-origin","sec-fetch-mode":"no-cors","sec-fetch-dest":"i
mage","referer":"http://localhost:3000/10","accept-encoding":"gzip, deflate, br","accept-language":"en-US,en;q=0.9,fr-FR;q=0.8,fr;q=0.7,ar-LB;q=0.6,ar;q=0.5",
"cookie":"Phpstorm-765b08c7=43941c52-b30c-4f77-8c00-ad2d1fd46856; Phpstorm-765b08c9=aad16ba7-2a86-4163-acab-0b7e3e830cd7; Webstorm-6c2b712d=264135ad-20fc-4499
-adeb-1848af7a3c65; connect.sid=s%3Alkk0ag7S0VF6nQg9AiHYRzONTe-VQ675.Co%2Brr0FvCWvMAhEvr1iDlSFcJkX3Q67tp5d7NR5qe%2BE; SESSIONID=s%3AjD-sbDcTAGaF8NUB6tmNz0i50X
MJuv6W.ZINajfxFnxhjG%2BaYK2LLLtvPl%2FjRny9XwdEDfyo8elc","if-none-match":"W/\"3-9/2caPgErNpmXSqwgiF7sVgzGPI\""},"remoteAddress":"::1","remotePort":35194},"msg"
:"Générer un nombre aléatoire"}
{"level":30,"time":1639356184249,"pid":6093,"hostname":"joseph-HP-EliteBook","req":{"id":2,"method":"GET","url":"/favicon.ico","query":{},"params":{},"headers
":{"host":"localhost:3000","connection":"keep-alive","sec-ch-ua":"\" Not A;Brand\";v=\"99\", \"Chromium\";v=\"96\", \"Google Chrome\";v=\"96\"","sec-ch-ua-mob
ile":"?0","user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.45 Safari/537.36","sec-ch-ua-platform":"\"Linu
x\"","accept":"image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8","sec-fetch-site":"same-origin","sec-fetch-mode":"no-cors","sec-fetch-dest":"i
mage","referer":"http://localhost:3000/10","accept-encoding":"gzip, deflate, br","accept-language":"en-US,en;q=0.9,fr-FR;q=0.8,fr;q=0.7,ar-LB;q=0.6,ar;q=0.5",
"cookie":"Phpstorm-765b08c7=43941c52-b30c-4f77-8c00-ad2d1fd46856; Phpstorm-765b08c9=aad16ba7-2a86-4163-acab-0b7e3e830cd7; Webstorm-6c2b712d=264135ad-20fc-4499
```

express-pino-logger est un middleware qui permet la journalisation **Pino** sur notre serveur Web Express.js. Nous les importons indépendamment afin de pouvoir interagir avec le pino logger à la fois directement et via notre middleware.

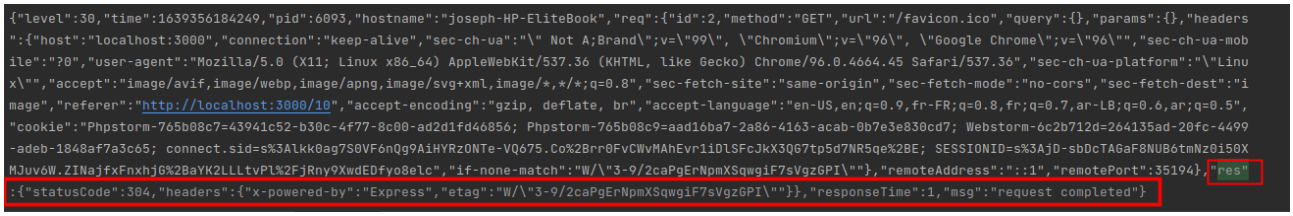
L'interface **Pino** est basée sur **Log4j**. **Log4j** est un enregistreur (logger) Apache écrit pour Java, mais des interprétations de son interface ont été implémentées dans de nombreux langages.

Pino vous permet de regrouper vos messages d'erreur par niveau. Les niveaux sont **trace**, **debug**, **info**, **warn**, **error** et **fatal**. Le niveau de journalisation par défaut est défini sur **info**.

Dans cet exemple, nous avons ajouté des messages de journal au niveau des informations. Pour ajouter le message de journal "Serveur écoute sur le port 3000", nous avons interagi directement avec **pino** à l'aide de **pino.info()**. Nous avons enregistré le middleware **express-pino-logger** à l'aide de la fonction **app.use()**.

Le middleware **express-pino-logger** ajoute un objet de journal à chaque requête entrante. Chaque message de journal est accessible via une propriété nommée log sur l'objet de requête (**req.log**). Chaque objet de journal est unique par requête. L'objet de journal contient des données sur la requête et un identifiant généré unique. L'identifiant unique généré permet de retracer les messages du journal jusqu'à une requête client spécifique.

Le middleware **express-pino-logger** génère également un message de journal pour chaque requête terminée. Une clé **res** est ajoutée au JSON. La clé **res** contient des données sur la réponse de la requête, y compris le code d'état et les en-têtes. Voici un exemple de message de journal de requête terminé:



Activation des journaux de débogage

debug est une bibliothèque populaire, utilisée par de nombreux frameworks notables, notamment les frameworks Web E**xpress.js** et **Koa.js** et le framework de test **Mocha**. **debug** est un petit utilitaire de débogage JavaScript basé sur la technique de débogage utilisée dans le noyau Node.js.

Dans cet exemple, nous allons découvrir comment activer les journaux de débogage sur une application Express.js.

Créez un fichier nommé **server_debug.js**:

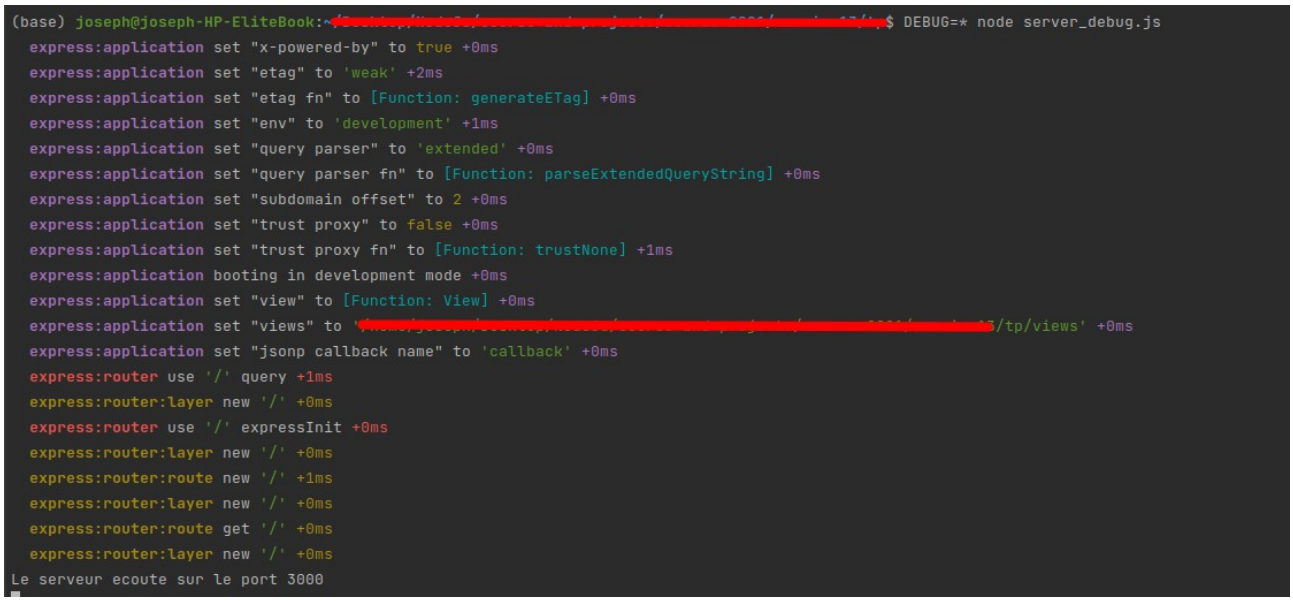
$ touch server_debug.js

Ajoutez le code suivant à server_debug.js:

```
const express = require("express");
const app = express();
app.get("/", (req, res) => res.send("Hello World!"));
app.listen(3000, () => {
      console.log("Le serveur ecoute sur le port 3000");
});
```

Pour activer la journalisation de débogage, démarrez votre serveur avec la commande suivante:

$ DEBUG=* node server_debug.js



Accédez à http://localhost:3000 dans votre navigateur pour envoyer une requête à notre serveur. Vous devriez voir que les messages de journal décrivant votre requête ont été affichés:

Arrêtez votre serveur en utilisant Ctrl + C. Maintenant, nous pouvons également filtrer les journaux de débogage générés. Nous allons les filtrer pour voir uniquement les actions du routeur Express.js. Pour cela, redémarrez votre serveur avec la commande suivante:

$ DEBUG=express:router* node server_debug.js

Attendez-vous à voir la sortie suivante dans votre fenêtre de terminal. Notez que seules les actions du routeur Express.js sont générées:

Nous commençons par ajouter DEBUG=* à notre commande de démarrage. Cette syntaxe transmet une variable d'environnement nommée DEBUG à notre processus Node.js, accessible depuis l'application via **process.env.DEBUG**.

Nous définissons la valeur sur *, ce qui active tous les journaux. Plus tard, nous filtrons les journaux en définissant **DEBUG=express:router***. En interne, le module de débogage convertit les valeurs que nous avons définies en expressions régulières.

Express.js utilise le module de débogage en interne pour instrumenter son code.

Utilisation du débogage en production:

La configuration de débogage par défaut n'est pas adaptée à la connexion en production. Les journaux de débogage par défaut sont destinés à être lisibles par l'homme, d'où le codage couleur. En production, vous devez transmettre à votre processus la valeur **DEBUG_COLORS=no** pour supprimer les codes ANSI qui implémentent le codage couleur. Cela rendra la sortie plus facilement lisible par machine.

Activation des journaux de débogage “core” Node.js

Lors du débogage de certains problèmes dans vos applications, il peut être utile d'avoir un aperçu des éléments internes de Node.js et de la façon dont il gère l'exécution de votre programme. Node.js fournit des journaux de débogage que nous pouvons activer pour nous aider à comprendre ce qui se passe en interne dans Node.js.

Ces journaux de débogage principaux peuvent être activés via une variable d'environnement nommée **NODE_DEBUG**. Dans cet exemple, nous allons définir la variable d'environnement **NODE_DEBUG** pour nous permettre de consigner les comportements internes de Node.js.

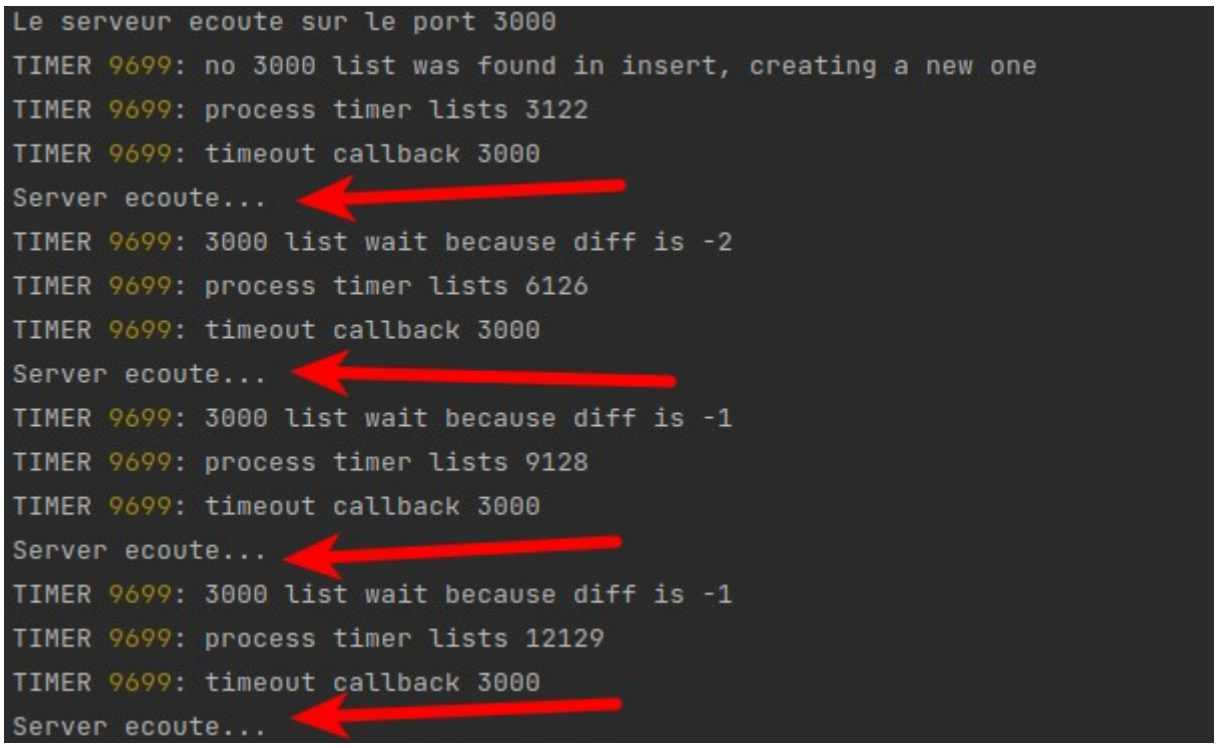
Remplacez le contenu de **server_debug.js** par ce qui suit:

```
const express = require("express");
const app = express();
app.get("/", (req, res) => {
      res.send("Hello World!");
});
app.listen(3000, () => {
      console.log("Le serveur ecoute sur le port 3000");
      setInterval(() => {
            console.log("Server ecoute...");
      }, 3000);
});
```

Maintenant que nous avons une application prête, nous pouvons activer les journaux de débogage “core” pour nous permettre de voir ce qui se passe au niveau de l'exécution de Node.js. Nous avons juste besoin de définir la variable **NODE_DEBUG** sur le “flag” **internal** que nous souhaitons enregistrer. Les indicateurs internes s'alignent sur des sous-systèmes spécifiques de Node.js, tels que les *timers* ou HTTP. Pour activer les journaux de débogage du *timer*, démarrez votre serveur avec la commande suivante:

$ NODE_DEBUG=timer node server_debug.js

Observez la sortie de journal supplémentaire de notre programme. Nous pouvons voir des informations supplémentaires sur notre fonction **setInterval()**, qui est exécutée toutes les 3000 ms:



Les instructions de journal TIMER précédentes sont des informations de débogage supplémentaires qui dérivent de l'implémentation interne des minuteurs dans le noyau Node.js, qui peuvent être trouvées à l'adresse **https://github.com/nodejs/node/blob/master/lib/internal/timers.js.**

Nous allons maintenant activer les journaux de débogage “core” pour le module http. Redémarrez votre serveur avec la commande suivante:

$ NODE_DEBUG=http node server_debug.js

Accédez à http://localhost:3000 dans un navigateur. Vous devez vous attendre à voir des journaux internes concernant la sortie de votre requête HTTP:

Dans cet exemple, nous définissons la variable d'environnement **NODE_DEBUG** sur les sous-systèmes **timer** et **http**. La variable d'environnement **NODE_DEBUG** peut être définie sur les sous-systèmes Node.js suivants:

child_process	http	net	stream
cluster	https	policy	timer
esm	http2	repl	tls
fs	module	source_map	worker

Il est également possible d'activer les journaux de débogage sur plusieurs sous-systèmes via la variable d'environnement **NODE_DEBUG**. Pour activer plusieurs journaux de sous-système, vous pouvez les transmettre sous forme de liste séparée par des virgules. Par exemple, pour activer à la fois les sous-systèmes **http** et **timer**, vous devez fournir la commande suivante:

$ NODE_DEBUG=http,timer node server_debug.js

La sortie de chaque message de journal comprend le sous-système, suivi de l'identificateur de processus (PID), puis du message de journal.

Augmentation de la taille de la trace de la pile (stack trace)

Une trace de pile, parfois appelée “stack backtrace” est définie comme une liste de trames de pile. Lorsque votre processus Node.js rencontre une erreur, une trace de pile s'affiche, détaillant la fonction qui a rencontré l'erreur et les fonctions par lesquelles elle a été appelée. Par défaut, le moteur V8 de Node.js renverra 10 trames de pile.

Lors du débogage de certaines erreurs, il peut être utile d'avoir plus de 10 trames de pile. Le nombre de trames de pile stockées a un coût en termes de performances. Le suivi des trames de pile supplémentaires entraînera une consommation de mémoire et de processeur plus importante par nos applications.

Dans cet exemple, nous allons augmenter la taille de la trace de la pile.

Ajoutez les nouveaux fichiers suivants pour cet exemple:

$ touch server_stacktrace.js routes.js

Ajoutez ce qui suit à “server_stacktrace.js”:

```
const express = require("express");
const routes = require("./routes");
const app = express();
app.use(routes);
app.listen(3000, () => {
	console.log("Le serveur ecoute sur le port 3000");
});
```

Et puis ajoutez ce qui suit à **routes.js**:

```
const express = require("express");
const router = new express.Router();
router.get("/", (req, res) => {
	res.send(recursiveContent());
});
function recursiveContent(content, i = 10) {
	--i;
	if (i !== 0) {
		return recursiveContent(content, i);
	} else {
		return content.undefined_property;
	}
}
module.exports = router;
```

Le but de la fonction recursiveContent() est de forcer la création d'appels de fonction, mais dans les applications plus volumineuses et plus complexes, il est possible de dépasser naturellement la limite des trames de pile (stack frames). Nous avons maintenant une application qui dépassera la limite de pile d'appels par défaut.

Commencez par exécuter le serveur:

$ node server_stacktrace.js

Maintenant, dans un navigateur, accédez à http://localhost:3000. Vous pouvez également utiliser cURL pour envoyer une requête au point de terminaison.

Observez que nous voyons la sortie de trace de pile suivante renvoyée:

```
Le serveur ecoute sur le port 3000
TypeError: Cannot read property 'undefined_property' of undefined
    at recursiveContent ([redacted]/routes.js:11:24)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
    at recursiveContent ([redacted]/routes.js:9:16)
```

Nous pouvons maintenant redémarrer notre application avec l'indicateur **--stack-trace-limit**. Nous allons définir ceci à 20:

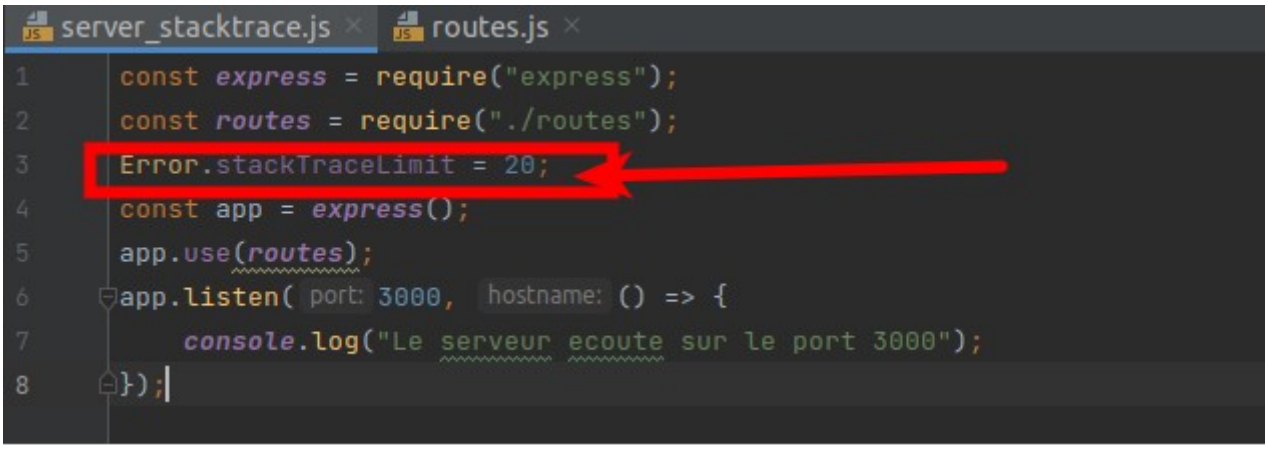
```
$ node --stack-trace-limit=20 server_stacktrace.js
```

Maintenant, naviguez ou envoyez à nouveau une requête à http://localhost:3000. Observez que nous avons maintenant plus de trames de la trace de la pile.

En étendant le nombre de trames de pile renvoyées, nous pouvons voir que la fonction **recursiveContent()** est appelée dans **routes.js**. Cela nous aide à comprendre pourquoi notre programme plante, c'est que nous n'avons pas défini "content" et nous le passons à notre fonction **recursiveContent()**.

Dans cet exemple, nous utilisons l'indicateur **--stack-trace-limit.** Cet indicateur indique au moteur JavaScript V8 de conserver davantage de piles. Lorsqu'une erreur se produit, la trace de la pile affichera les appels de fonction précédents jusqu'à la limite définie avec le drapeau. Dans cet exemple, nous avons étendu cela à 20 trames de pile.

Notez qu'il est également possible de définir cette limite à partir de votre code d'application. La ligne suivante définirait la limite de trace de pile à 20:



Il est également possible de définir la limite de trace de la pile sur **Infinity**, ce qui signifie que tous les appels de fonction précédents seront conservés:

```
Error.stackTraceLimit = Infinity;
```

Le stockage de traces de pile supplémentaires a un coût en termes de performances en termes d'utilisation du processeur et de la mémoire. Vous devriez considérer l'impact que cela peut avoir sur votre application.