How to Debug a Node.js app in a Docker Container

In this post, we'll take a look at how you can debug a Node.js app in a Docker container.

> *In case you're wondering: "Why should I acquire this knowledge?" The answer is simple:*

Most of the time you can be well off running your app on your local machine and use containers only to sandbox your databases and messaging queues, but some bugs will show themselves only when the app itself is containerized as well. In these cases, it is very helpful to know how to attach a debugger to the service.

According to the Foundation's Node.js Developer Survey, half of Node.js users use Docker for development. While containerization, in general, is a very powerful tool - and here at RisingStack we always start new projects by spinning up the needed infrastructure in a docker-compose.yaml - it can be tricky to reach the enveloped Node process if you don't know how to do it.

> *In case you need guidance with Docker, Kubernetes, Microservices or Node.js, feel free to ping us at info@risingstack.com!*

All code snippets and settings used in this post can be found in its dedicated GitHub repo.

How to use the Node inspector

If you mostly use printf, aka caveman debugging, it can be very difficult to find the right value at the right time.

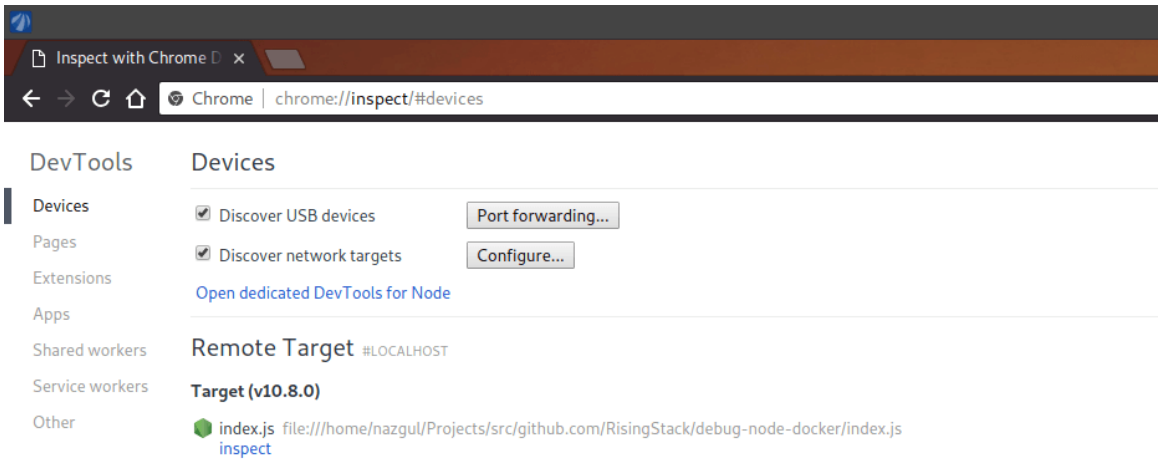
Things get even worse if you have to rebuild your container image each time you add console.log to it. It could be a lot easier to have the image built once and jump around within it, examining your variables while it's running. To better understand what we're gonna do here, I highly suggest to familiarize yourself with the node inspect commands first.

To run your Node app in debug mode, simply add inspect after node, something like that:

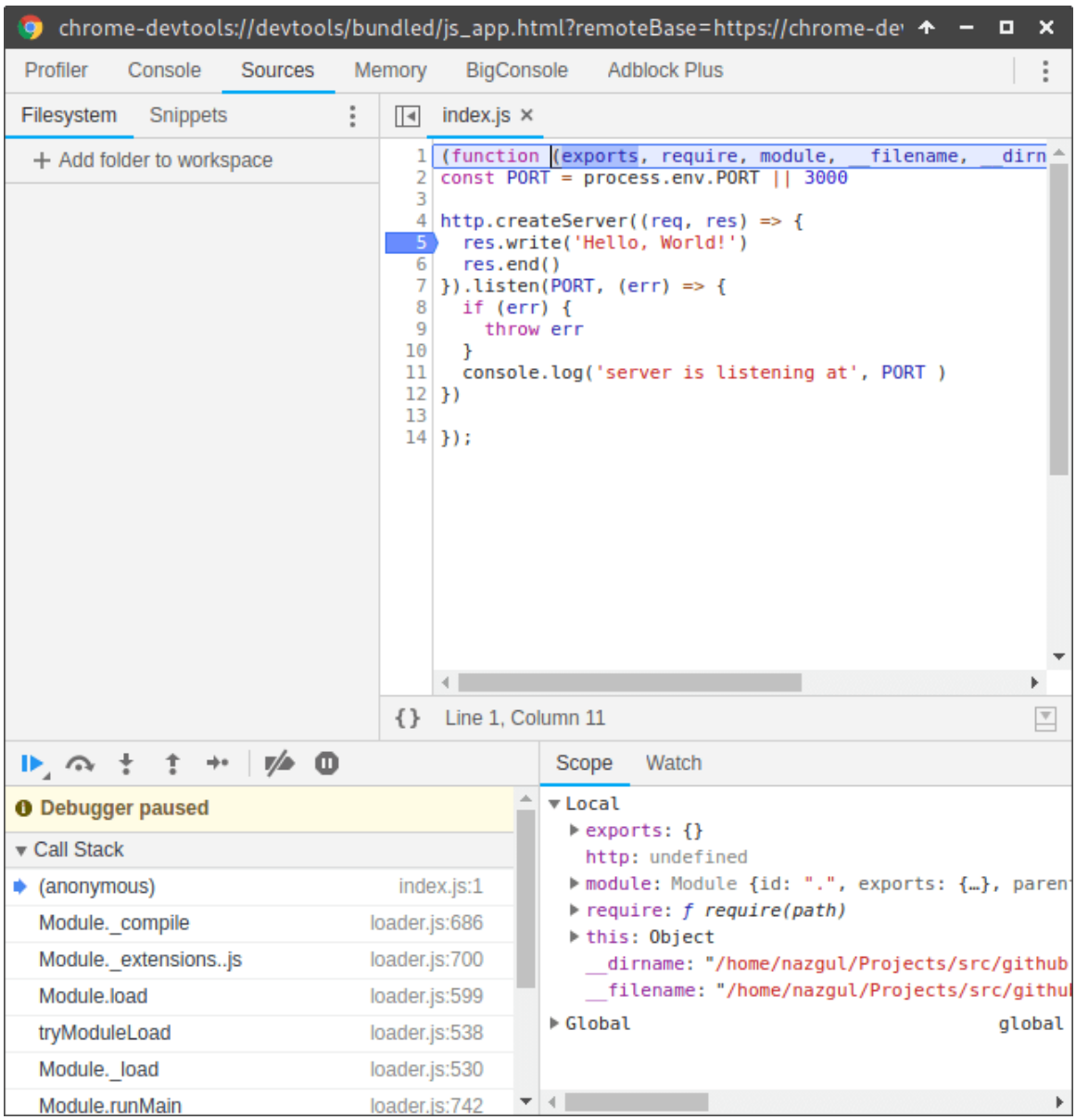
```
$ node inspect index.js
< Debugger listening on ws://127.0.0.1:9229/5adb6217-0757-4761-95a2-6af0955d7d25
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in index.js:1
> 1 (function (exports, require, module, __filename, __dirname) { const http = require('http')
  2 const PORT = process.env.PORT || 3000
  3
debug>
```

When you run your code in inspect mode, it always stops at the first line, waiting for you to interact with it. For those who were brought up using gdb to debug their code, this interface might be compelling. However, if you are used to interacting with your debugger using a GUI, you might want to open up your chrome and navigate to chrome://inspect.

You should see something like this:



Under remote target, click inspect and you'll be presented with the Chrome Developer Tools debugger.



Now you can use the debugger as you please. It's time to wrap our app in a container.

Debugging Node.js in a Docker container

First, we'll need to create a Dockerfile,

```
FROM node

COPY package.json package.json
RUN npm install

COPY . .

EXPOSE 3000
CMD ["node", "."]
```

and a `docker-compose.yaml`

```
version: '3.6'

services:
  app:
    build: .
    ports:
      - "3000:3000"
```

Now if you run `docker-compose up`, you'll be able to reach your service on `http://localhost:3000`.

The next step is to expose the debug port to the outside world. First, let's create a `debug-compose.yaml`.

```
version: '3.6'

services:
  app:
```

```
  build: .
  ports:
    - "3000:3000"
    - "9229:9229"
  command:
    - node
    - "--inspect-brk=0.0.0.0"
    - "."
```

As you can see, we opened up port 9229, which is the debug port of Node.js apps. We also overrode the command we specified in the Dockerfile. The `--inspect-brk=0.0.0.0` argument does two different things:

1. `--inspect` tells Node that we want to run our app in debug mode.
2. by adding `-brk` we also make sure that the app stops at the first line, so we have enough time to open up the inspector
3. adding `=0.0.0.0` opens up the debugger to connections from any IP.

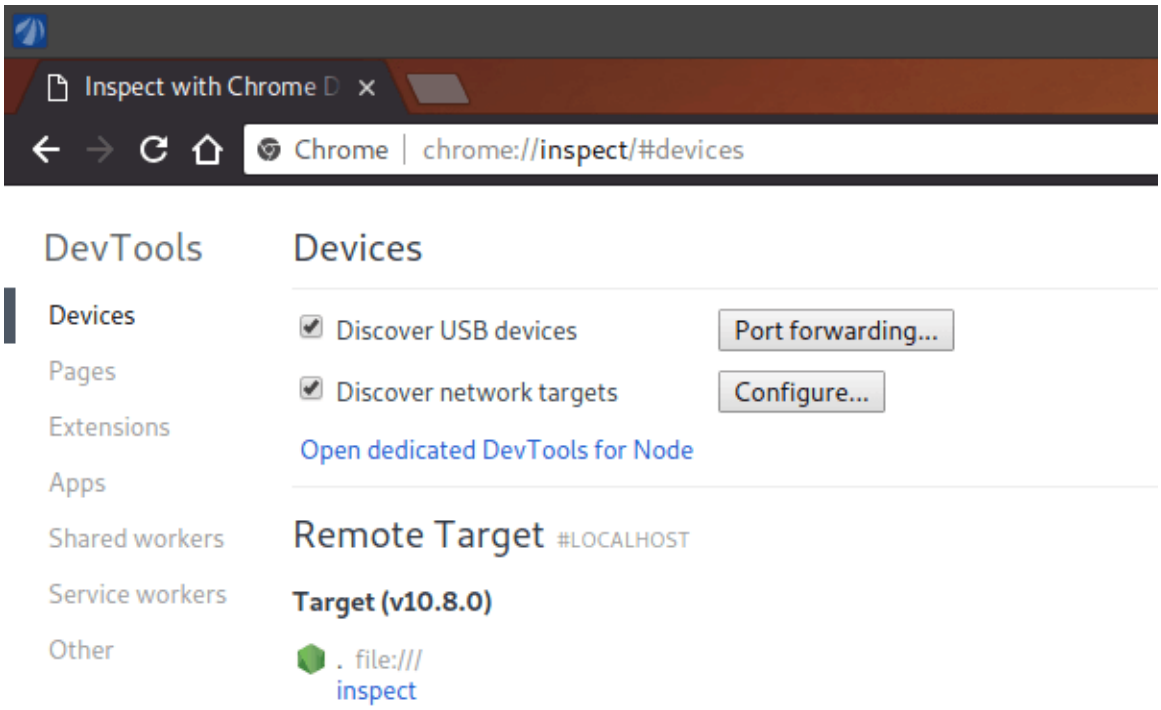
By default, the inspector is bound to `127.0.0.1` which makes sense, as we usually don't want to allow people from all around the world to attach a debugger to our app. However, the container is a different host with a different IP than our host machine, so we won't be able to reach it. It is fine as long as we do it locally; however, we don't want to run it on a live server like this.

For this reason **make sure it is a different file from your `docker-compose.yaml`**.

With a bit more work, you can expose the debug port from your staging cluster to your IP — but in that case, to **your IP only** — and debug issues there as well.

Also, note that the port forwarding rules are enclosed in `"`-s. If you omit the quotes the rule might not work, making it difficult to figure out why you're unable to attach the debugger to your process.

With all that said, you should be able to inspect your app in the dev tools.

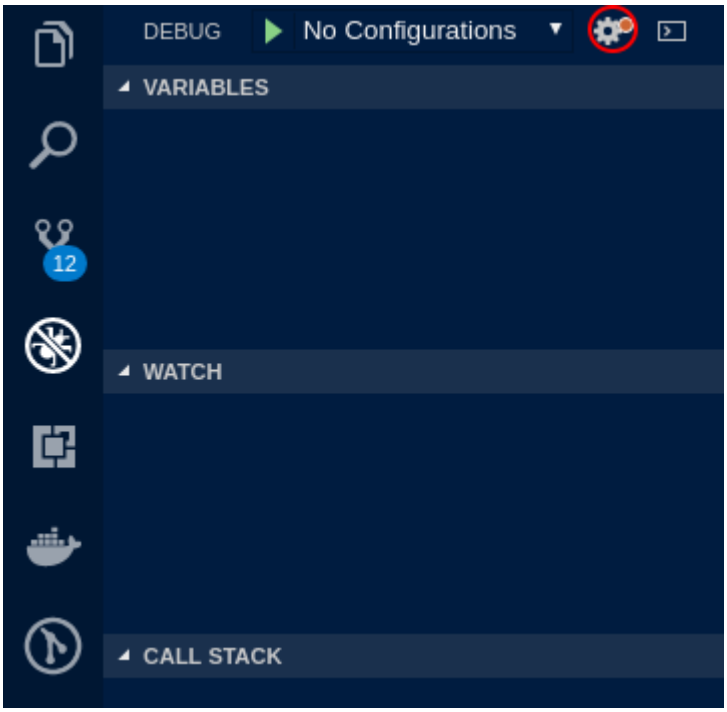


Debugging with Visual Studio Code

It is great to use the inspector for single file issues, though it can have problems discovering all the files in your project. In these cases, it's better to attach the debugger provided by your IDE. Let's see how it's done with Visual Studio Code.

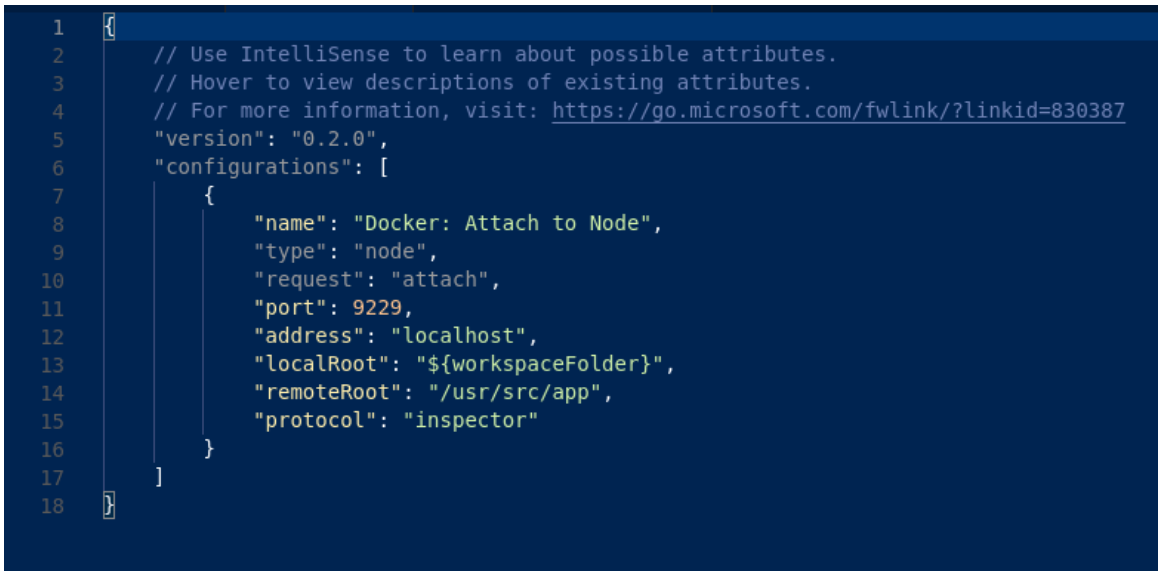
First, navigate to the debug tab

then click the gear icon



from the popup list, select docker (make sure, you have the Docker extension installed)

it should generate a launch.json in the projects .vscode folder that looks like this:



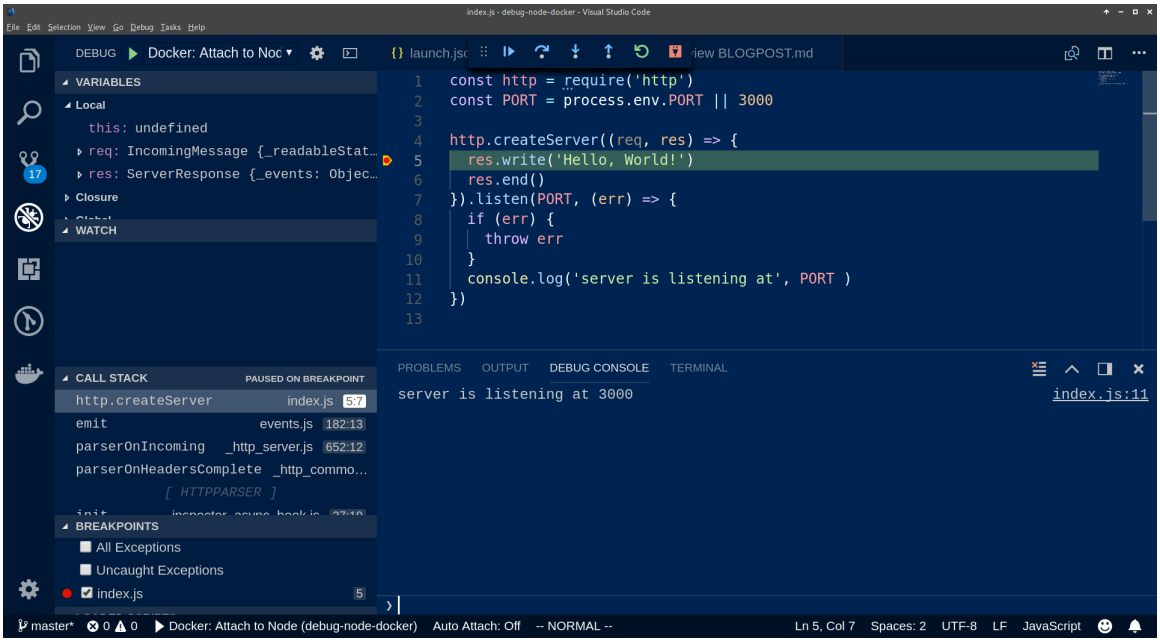
```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Docker: Attach to Node",
            "type": "node",
            "request": "attach",
            "port": 9229,
            "address": "localhost",
            "localRoot": "${workspaceFolder}",
            "remoteRoot": "/usr/src/app",
            "protocol": "inspector"
        }
    ]
}
```

It's almost ok, though in our case, the root of our app is the root of the container's filesystem, so we need to update that as well. The object should look like this when you're done:

```
{
  "name": "Docker: Attach to Node",
  "type": "node",
  "request": "attach",
  "port": 9229,
  "address": "localhost",
  "localRoot": "${workspaceFolder}",
  "remoteRoot": "/",
  "protocol": "inspector"
}
```

Now, if you hit F5 on your keyboard, you'll be prompted with the debugger you got used to in VSCode. Hit F5 again, to let the server start listening. If you put a

breakpoint somewhere and call the server at http://localhost:3000 you should see this



Why not ndb?

Even though ndb is great for debugging, you cannot attach it to running processes currently, which pretty much ruins the purpose in our case.

You could also start your debug process with it inside the container and attach your debugger to if from outside, but you also need to modify your Dockerfile to do so, and you don't really gain anything as you would need to attach your chrome vscode, or another debugger to it anyway. Follow this issue for updates on the matter.

Final thoughts on Node.js Debugging

Seeing how container technologies such as Kubernetes, AWS ECS, Docker Swarm and others are getting more and more widespread it is clearly visible that containers are here to stay.

> *In case you need guidance with Docker, Kubernetes, Microservices or Node.js, feel free to ping us at info@risingstack.com!*

The fact that you can have the same image run on your local machine while you're developing that will eventually land on the cluster is definitely a nice thing

as you can bundle the app with the configuration and deploy them together. However, finding bugs that only show themselves when the app is bundled up can be difficult when you rely on printf debugging, so even if you have not used it so far, it is definitely a good idea to become friends with debuggers and learn how to attach them to processes running in your containers.

Happy debugging!

The idea for this post came when we ran into a bug that only arose in the container with @fazekasda. Thanks for the help man!